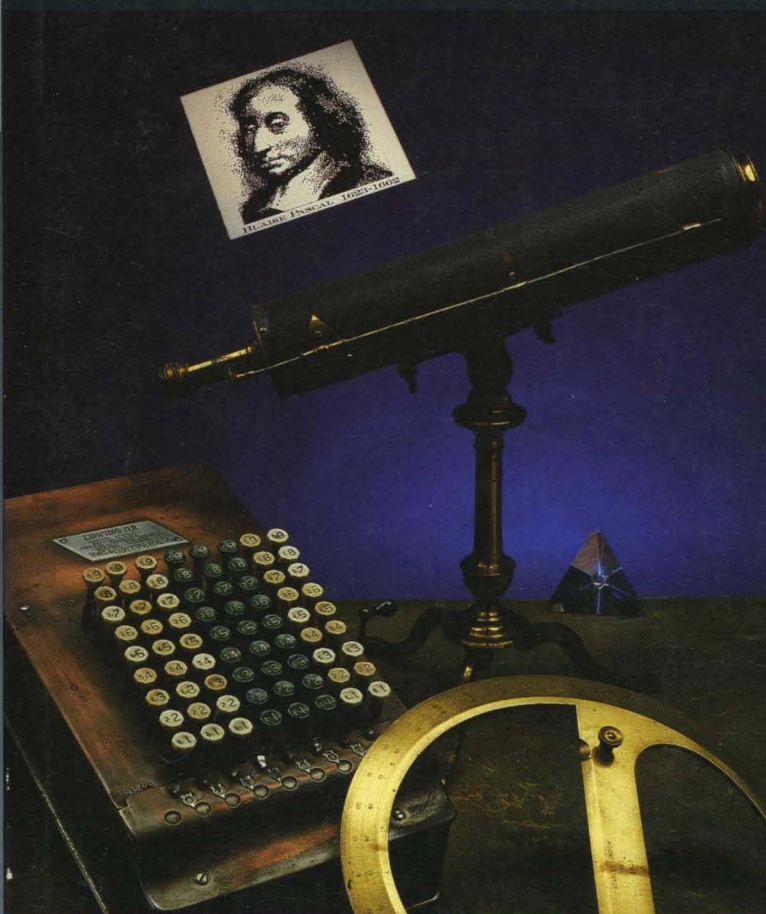


TURBO PASCAL TOOLBOX NUMERICAL METHODS

*A complete collection
of Turbo Pascal®
routines and programs*

*Provides state-of-
the-art math tools
to solve scientific
and engineering
problems—
fast!*



IBM® VERSION

PC, XT,® AT,® & True Compatibles



Turbo Pascal Numerical Methods Toolbox

Borland's No-Nonsense License Statement!

This software is protected by both United States copyright law and international treaty provisions. Therefore, you must treat this software *just like a book*, with the following single exception. Borland International authorizes you to make archival copies of the software for the sole purpose of backing-up our software and protecting your investment from loss.

By saying, "just like a book," Borland means, for example, that this software may be used by any number of people and may be freely moved from one computer location to another, so long as there is **no possibility** of it being used at one location while it's being used at another. Just like a book that can't be read by two different people in two different places at the same time, neither can the software be used by two different people in two different places at the same time. (Unless, of course, Borland's copyright has been violated.)

Borland International grants you (the licensed owner of the Turbo Pascal Numerical Toolbox) the right to incorporate toolbox routines into your programs. You may distribute your programs that contain Numerical Toolbox routines in executable form without restriction or fee, but you may not give away or sell any part of the actual Numerical Methods Toolbox source code. You are not, of course, restricted from distributing your own source code.

Sample programs are provided on the Numerical Methods Toolbox diskettes as examples of how to use the various toolbox features. You may edit or modify these sample programs and incorporate them into the programs that you write. Use of these sample programs is governed by the same conditions and restrictions as outlined in the first paragraph above.

WARRANTY

With respect to the physical diskette and physical documentation enclosed herein, Borland International, Inc. ("Borland") warrants the same to be free of defects in materials and workmanship for a period of 60 days from the date of purchase. In the event of notification within the warranty period of defects in material or workmanship, Borland will replace the defective diskette or documentation. **If you need to return a product, call the Borland Customer Service Department to obtain a return authorization number.** The remedy for breach of this warranty shall be limited to replacement and shall not encompass any other damages, including but not limited to loss of profit, and special, incidental, consequential, or other similar claims.

Borland International, Inc. specifically disclaims all other warranties, expressed or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose with respect to defects in the diskette and documentation, and the program license granted herein in particular, and without limiting operation of the program license with respect to any particular application, use, or purpose. In no event shall Borland be liable for any loss of profit or any other commercial damage, including but not limited to special, incidental, consequential or other damages.

GOVERNING LAW

This statement shall be construed, interpreted, and governed by the laws of the state of California.

Turbo Pascal

Numerical Methods Toolbox™

Copyright ©1986
All Rights Reserved
BORLAND INTERNATIONAL, INC.
4585 SCOTTS VALLEY DRIVE
SCOTTS VALLEY, CALIFORNIA 95066
USA

Table of Contents

Introduction	1
Toolbox Functions	1
About this Manual	2
On the Distribution Disks	3
System Requirements	3
Acknowledgements	4
 Chapter 1. ROUTINE BEGINNINGS	 5
Using the Toolbox: An Example	5
The Distribution Disks	7
Installation	8
The Graphics Demos	12
Data Types and Defined Constants	12
Compiler Directives	13
 Chapter 2. ROOTS TO EQUATIONS IN ONE VARIABLE	 15
Stopping Criteria	17
Root of a Function Using the Bisection Method (BISECT.INC)	18
Description	18
User-Defined Function	18
Input Parameters	18
Output Parameters	19
Syntax of the Procedure Call	19

Comments	19
Sample Program	19
Example	20
Root of a Function Using the Newton-Raphson Method (RAPHSON.INC)	21
Description	21
User-Defined Functions	21
Input Parameters	21
Output Parameters	22
Syntax of the Procedure Call	22
Comments	22
Sample Program	22
Example	23
Root of a Function Using the Secant Method (SECANT.INC)	25
Description	25
User-Defined Function	25
Input Parameters	25
Output Parameters	26
Syntax of the Procedure Call	26
Comments	26
Sample Program	26
Example	27
Real Roots of a Real Polynomial Equation Using the Newton-Horner Method with Deflation (NEWTDEFL.INC)	28
Description	28
User-Defined Types	28
Input Parameters	28
Output Parameters	29
Syntax of the Procedure Call	30
Comments	30
Sample Program	30
Input Files	30
Example	30
Complex Roots of a Complex Function Using Müller's Method (MULLER.INC)	33
Description	33
User-Defined Types	33
User-Defined Procedure	33
Input Parameters	33
Output Parameters	34
Syntax of the Procedure Call	34
Comments	34
Sample Program	35
Example	35

Complex Roots of a Complex Polynomial Using Laguerre's Method and Deflation (LAGUERRE.INC)	37
Description	37
User-Defined Types	37
Input Parameters	37
Output Parameters	38
Syntax of the Procedure Call	38
Comments	38
Sample Program	39
Input Files	39
Example	39
Chapter 3. INTERPOLATION	43
Polynomial Interpolation Using Lagrange's Method (LAGRANGE.INC)	45
Description	45
User-Defined Types	45
Input Parameters	45
Output Parameters	46
Syntax of the Procedure Call	46
Sample Program	46
Input Files	46
Example	47
Interpolation Using Newton's Interpolary Divided-Difference Method (DIVDIF.INC)	49
Description	49
User-Defined Types	49
Input Parameters	49
Output Parameters	50
Syntax of the Procedure Call	50
Sample Program	50
Input Files	50
Example	50
Free Cubic Spline Interpolation (CUBE_FRE.INC)	52
Description	52
User-Defined Types	52
Input Parameters	52
Output Parameters	53
Syntax of the Procedure Call	53
Sample Program	53
Input Files	54
Example	54
Clamped Cubic Spline Interpolation (CUBE_CLA.INC)	57
Description	57
User-Defined Types	57

Input Parameters	57
Output Parameters	58
Syntax of the Procedure Call	58
Sample Program	59
Input Files	59
Example	59
Chapter 4. NUMERICAL DIFFERENTIATION	63
First Differentiation Using Two-Point, Three-Point, or Five-Point Formulas	
(DERIV.INC)	66
Description	66
User-Defined Types	66
Input Parameters	66
Output Parameters	67
Syntax of the Procedure Call	67
Comments	67
Sample Program	68
Input Files	68
Example	68
Second Differentiation Using Three-Point or Five-Point Formulas	
(DERIV2.INC)	71
Description	71
User-Defined Types	71
Input Parameters	71
Output Parameters	72
Syntax of the Procedure Call	72
Comments	72
Sample Program	73
Input Files	73
Example	73
Differentiation with a Cubic Spline Interpolant (INTERDRV.INC)	76
Description	76
User-Defined Types	76
Input Parameters	76
Output Parameters	77
Syntax of the Procedure Call	77
Sample Program	77
Input Files	78
Example	78
Differentiation of a User-Defined Function (DERIVFN.INC)	80
Description	80
User-Defined Types	80
User-Defined Function	80
Input Parameters	80

Output Parameters	81
Syntax of the Procedure Call	81
Comments	81
Sample Program	81
Input Files	81
Example	82
Second Differentiation of a User-Defined Function (DERIV2FN.INC)	83
Description	83
User-Defined Types	83
User-Defined Function	83
Input Parameters	83
Output Parameters	84
Syntax of the Procedure Call	84
Comments	84
Sample Program	84
Input Files	85
Example	85
Chapter 5. NUMERICAL INTEGRATION	87
Integration Using Simpson's Composite Algorithm (SIMPSON.INC)	89
Description	89
User-Defined Function	89
Input Parameters	89
Output Parameters	90
Syntax of the Procedure Call	90
Sample Program	90
Example	90
Integration Using the Trapezoid Composite Rule (TRAPZOID.INC)	92
Description	92
User-Defined Function	92
Input Parameters	92
Output Parameters	93
Syntax of the Procedure Call	93
Sample Program	93
Example	93
Integration Using Adaptive Quadrature and Simpson's Rule (ADAPSIMP.INC)	95
Description	95
User-Defined Function	95
Input Parameters	95
Output Parameters	96
Syntax of the Procedure Call	96
Comments	96
Sample Program	96

Example	97
Integration Using Adaptive Quadrature and Gaussian Quadrature (ADAPGAUS.INC)	98
Description	98
User-Defined Function	98
Input Parameters	98
Output Parameters	99
Syntax of the Procedure Call	99
Comments	99
Sample Program	101
Example	101
Integration Using the Romberg Algorithm (ROMBERG.INC)	102
Description	102
User-Defined Function	102
Input Parameters	102
Output Parameters	103
Syntax of the Procedure Call	103
Sample Program	103
Example	103
Chapter 6. MATRIX ROUTINES	105
Determinant of a Matrix (DET.INC)	107
Description	107
User-Defined Types	107
Input Parameters	107
Output Parameters	108
Syntax of the Procedure Call	108
Sample Program	108
Input File	108
Example	109
Inverse of a Matrix (INVERSE.INC)	110
Description	110
User-Defined Types	110
Input Parameters	110
Output Parameters	111
Syntax of the Procedure Call	111
Sample Program	111
Input Files	111
Example	112
Solving a System of Linear Equations with Gaussian Elimination (GAUSELIM.INC)	114
Description	114
User-Defined Types	114
Input Parameters	114

Output Parameters	115
Syntax of the Procedure Call	115
Sample Program	115
Input File	115
Example	116
Solving a System of Linear Equations with Gaussian Elimination and Partial Pivoting (PARTPIVT.INC)	117
Description	117
User-Defined Types	117
Input Parameters	117
Output Parameters	118
Syntax of the Procedure Call	118
Sample Program	118
Input File	118
Example	119
Solving a System of Linear Equations with Direct Factoring (DIRFACT.INC)	120
Description	120
User-Defined Types	120
Procedure LU_Decompose Input Parameters	121
Procedure LU_Decompose Output Parameters	121
Syntax of the Procedure Call	121
Procedure LU_Solve Input Parameters	121
Procedure LU_Solve Output Parameters	122
Syntax of the Procedure Call	122
Sample Program	122
Input File	122
Example	123
Solving a System of Linear Equations with the Iterative Gauss-Seidel Method (GAUSSIDL.INC)	126
Description	126
User-Defined Types	126
Input Parameters	127
Output Parameters	127
Syntax of the Procedure Call	128
Sample Program	128
Input File	128
Example	129
Chapter 7. EIGENVALUES AND EIGENVECTORS	131
Real Dominant Eigenvalue and Eigenvector of a Real Matrix Using the Power Method (POWER.INC)	133
Description	133
User-Defined Types	133

Input Parameters	133
Output Parameters	134
Syntax of the Procedure Call	134
Comments	134
Sample Program	135
Input File	135
Example	135
Real Eigenvalue and Eigenvector of a Real Matrix Using the Inverse	
Power Method (INVPOWER.INC)	137
Description	137
User-Defined Types	137
Input Parameters	137
Output Parameters	138
Syntax of the Procedure Call	138
Comments	139
Sample Program	139
Input File	139
Example	140
Real Eigenvalues and Eigenvectors of a Real Matrix Using the Power	
Method and Wielandt's Deflation (WIELANDT.INC)	143
Description	143
User-Defined Types	143
Input Parameters	143
Output Parameters	144
Syntax of the Procedure Call	145
Comments	145
Sample Program	146
Input File	146
Example	146
The Complete Eigensystem of a Symmetric Real Matrix Using the Cyclic	
Jacobi Method (JACOBI.INC)	149
Description	149
User-Defined Types	149
Input Parameters	149
Output Parameters	150
Syntax of the Procedure Call	150
Comments	151
Sample Program	151
Input File	151
Example	152
Chapter 8. INITIAL VALUE AND BOUNDARY VALUE METHODS	155
Solution to an Initial Value Problem for a First-Order Ordinary Differential	
Equation Using the Runge-Kutta Method (RUNGE_1.INC)	159

Description	159
User-Defined Types	159
User-Defined Function	160
Input Parameters	160
Output Parameters	160
Syntax of the Procedure Call	161
Comments	161
Sample Program	161
Example	162
Solution to an Initial Value Problem for a First-Order Ordinary Differential	
Equation Using the Runge-Kutta-Fehlberg Method (RKF_1.INC)	163
Description	163
User-Defined Types	163
User-Defined Function	163
Input Parameters	164
Output Parameters	164
Syntax of the Procedure Call	164
Comments	165
Sample Program	165
Example	166
Solution to an Initial Value Problem for a First-Order Ordinary Differential	
Equation Using the Adams-Bashforth/Adams-Moulton Predictor/Corrector	
Scheme (ADAMS_1.INC)	168
Description	168
User-Defined Types	169
User-Defined Function	169
Input Parameters	169
Output Parameters	169
Syntax of the Procedure Call	170
Comments	170
Sample Program	170
Example	171
Solution to an Initial Value Problem for a Second-Order Ordinary Differential	
Equation Using the Runge-Kutta Method (RUNGE_2.INC)	172
Description	172
User-Defined Types	173
User-Defined Function	173
Input Parameters	173
Output Parameters	174
Syntax of the Procedure Call	174
Comments	174
Sample Program	175
Example	175

Solution to an Initial Value Problem for an nth-Order Ordinary Differential Equation Using the Runge-Kutta Method (RUNGE_N.INC)	178
Description	178
User-Defined Types	180
User-Defined Function	180
Input Parameters	180
Output Parameters	181
Syntax of the Procedure Call	181
Comments	181
Sample Program	182
Example	182
Solution to an Initial Value Problem for a System of Coupled First-Order Ordinary Differential Equations Using the Runge-Kutta Method (RUNGE_S1.INC)	186
Description	186
User-Defined Types	188
User-Defined Functions	188
Input Parameters	189
Output Parameters	189
Syntax of the Procedure Call	190
Comments	190
Sample Program	191
Example	191
Solution to an Initial Value Problem for a System of Coupled Second-Order Ordinary Differential Equations Using the Runge-Kutta Method (RUNGE_S2.INC)	196
Description	196
User-Defined Types	199
User-Defined Functions	199
Input Parameters	200
Output Parameters	201
Syntax of the Procedure Call	201
Comments	201
Sample Program	203
Example	203
Solution to Boundary Value Problem for a Second-Order Ordinary Differential Equation Using the Shooting and Runge-Kutta Methods (SHOOT2.INC) ..	208
Description	208
User-Defined Types	209
User-Defined Function	209
Input Parameters	209
Output Parameters	210
Syntax of the Procedure Call	210
Comments	210

Sample Program	211
Example	211
Solution to a Boundary Value Problem for a Second-Order Ordinary Linear Differential Equation Using the Linear Shooting and Runge-Kutta Methods (LINSHOT2.INC)	215
Description	215
User-Defined Types	216
User-Defined Function	216
Input Parameters	216
Output Parameters	216
Syntax of the Procedure Call	217
Comments	217
Sample Program	218
Example	218
 Chapter 9. LEAST-SQUARES APPROXIMATION	221
Least-Squares Approximation (LEAST.INC)	222
Description	222
User-Defined Types	223
Input Parameters	223
Output Parameters	224
Syntax of the Procedure Call	224
Comments	224
POLY.LSQ	224
FOURIER.LSQ	225
POWER.LSQ	225
EXP.LSQ	225
LOG.LSQ	226
USER.LSQ	226
Sample Program	227
Input Files	227
Example	227
 Chapter 10. FAST FOURIER TRANSFORM ROUTINES	233
The Application Programs	235
Data Sampling	239
User-Defined Types	240
Fast Fourier Transform Algorithms	241
Procedure TestInput	241
Description	241
Input Parameters	241
Output Parameters	241
Syntax of the Procedure Call	241
Procedure MakeSinCosTable	242

Description	242
Input Parameters	242
Output Parameters	242
Syntax of the Procedure Call	242
Procedure FFT	242
Description	242
Input Parameters	243
Output Parameters	243
Syntax of the Procedure Call	243
Fast Fourier Transform Applications	244
COMPPFFT.INC	244
Description	244
Input Parameters	244
Output Parameters	244
Syntax of the Procedure Call	245
Comments	245
REALFFT.INC	245
Description	245
Input Parameters	245
Output Parameters	246
Syntax of the Procedure Call	246
Comments	246
COMPCNVL.INC	246
Description	246
Input Parameters	247
Output Parameters	247
Syntax of the Procedure Call	247
Comments	247
REALCNVL.INC	248
Description	248
Input Parameters	248
Output Parameters	248
Syntax of the Procedure Call	249
Comments	249
COMPCORR.INC	249
Description	249
Input Parameters	250
Output Parameters	250
Syntax of the Procedure Call	250
Comments	250
REALCORR.INC	251
Description	251
Input Parameters	251
Output Parameters	252

Syntax of the Procedure Call	252
Comments	252
Sample Program	252
Input File	253
Example	253
Chapter 11. GRAPHICS PROGRAMS	261
Function of the Least-Squares Graphics Demonstration Program	262
Function of the Fourier Transform Graphics Demonstration Program	264
Printing	265
Rebuilding LSQIBM.COM	267
Rebuilding FFTIBM.COM	268
Rebuilding for the Hercules Card	269
Rebuilding for the EGA Card	269
Using the Math Coprocessor	270
REFERENCES	271
INDEX	273

Introduction

The Turbo Numerical Methods Toolbox is a reference manual for both the student of numerical analysis and the professional needing efficient routines. An elementary background in calculus and linear algebra is assumed, although many of the algorithms use only high-school-level mathematics. A general knowledge of Turbo Pascal® is also assumed. If you need to brush up on your knowledge of Pascal, we suggest looking at the *Turbo Pascal Reference Manual* and/or the *Turbo Pascal Tutor Manual*.

Before you begin using a particular routine, read through this brief introductory chapter and then refer to the chapter that interests you.

Toolbox Functions

The Turbo Pascal Numerical Methods Toolbox provides routines for

- Finding solutions to equations
- Interpolations
- Calculus
- Numerical derivatives and integrals
- Matrix operations: inversions, determinants, and eigenvalues

- Differential equations
- Least-squares approximations
- Fourier transforms

About this Manual

The major areas in numerical analysis are represented in this Toolbox, with each chapter focusing on a particular problem. Each routine begins with a general description of the implemented algorithm or numerical method. (References to numerical analysis texts are provided for each numerical procedure.) User-supplied types, functions, and input and output parameters are defined, and the syntax of the procedure call is provided. If appropriate, a “Comments” section is also provided.

Finally, every algorithm in the Toolbox is accompanied by a general-purpose program that handles all the necessary I/O, while allowing you to try each algorithm without building any code. Handily, these sample programs will often reduce the coding your own application may require.

As an example, let’s say you want to find the roots to an equation in one variable. First, you would read the introduction to Chapter 2, “Roots to Equations in One Variable,” and choose the numerical method best suited to your particular problem. Second, you would run the sample program for the desired numerical method to determine the necessary input and output. Third, you would write a Turbo Pascal function defining your equation, using the function already coded in the sample program as a guide. Fourth, you would run the sample program with your function substituted for the original one. Of course, if these algorithms are to be part of a larger program, you must build all the interfaces to the other parts of the system; but this should only be done after you gain experience with the particular numerical method.

Several books are referred to throughout the text; complete references are listed at the back of the book in the section entitled “References.”

The body of this manual is printed in normal typeface; other typefaces serve to illustrate the following:

Alternate	This type displays program examples and procedure and function declarations.
<i>Italics</i>	This type emphasizes certain concepts, first-mentioned terms, and mathematical expressions.
Boldface	This type marks the reserved words of Turbo Pascal in text and in program examples.

On the Distribution Disks

The routines for this Toolbox are contained on three packed disks. Their contents and general installation instructions are covered in Chapter 1.

System Requirements

All routines will run in standard Turbo Pascal version 3.0. (They will also run in version 2.0, but you must make one change to use the sample programs; see the section entitled "Installation" in Chapter 1.) All sample programs will run on an IBM® PC or compatible machine using DOS 2.0 or greater.

A small portion of the Toolbox uses graphics (see Chapter 11). These programs are for PC-DOS users only, requiring either an IBM PC Color Graphics Adapter, an IBM Enhanced Graphics Adapter, or a Hercules Monochrome Graphics Adapter. Recompiling these requires the Turbo Pascal Graphix Toolbox® version 1.06A or later, and Turbo Pascal 3.0. They can also be recompiled for the EGA and several other cards using version 1.07A of the Graphix Toolbox.

We strongly recommend that anyone serious about numerical analysis invest in hardware and software to run Turbo Pascal with support for the Intel 8087 numerical processing chip:

- Hardware: an 8087 chip plugged into the motherboard of a PC, XT® or equivalent, or an 80287 chip in an AT® or equivalent.
- Software: TURBO-87.COM, the version of Turbo Pascal designed to take advantage of the 8087 chip.

For machines running the Intel 8088 CPU, the increase in execution speed of programs using real-number arithmetic is often a factor of ten or more, while for 80286 machines, the increase is only about a factor of two (Fried 1985). Perhaps more important than speed is the increase in accuracy — 16 significant figures accuracy for Turbo Pascal with 8087 support versus 11 significant figures for standard Turbo. Since round-off errors are a serious concern in numerical analysis, the increased accuracy is of great value.

All of the examples in this manual were run using Turbo-87. If you run them without the Turbo-87, you will usually get less accurate answers.

Acknowledgements

We refer to several products throughout the manual; following is a list of each one and its respective company.

- Turbo Pascal, Turbo Pascal Graphix Toolbox, SideKick, SuperKey, and Reflex: The Database Manager are registered trademarks and Turbo Pascal Numerical Methods Toolbox is a trademark of Borland International, Inc.
- IBM, XT, and AT are registered trademarks of International Business Machines Corp.
- MS-DOS is a registered trademark of Microsoft Corp.

Routine Beginnings

This chapter provides you with everything you need to start using the routines in this Toolbox. We'll discuss how to unpack the disks for use and list the files available once the disks are unpacked. We also briefly discuss data types and defined constants used in the Toolbox, and the setting of compiler directives.

First, though, before we thrust you into the middle of numerical madness, let's take a look at one way to use this Toolbox.

Using the Toolbox: An Example

In late 1986 and early 1987, the America's Cup 12-meter yacht championship was held. The 12-meter yachts are just large sailboats, but the competition is so intense that the only way to be competitive is to use dozens of people, spend millions of dollars, design a special boat, and spend a couple of years training for the race. The race has become so sophisticated that many of the sailboats have on-board computers and other electronic equipment.

To keep stride with other challengers, one yacht's crew used personal computers, and of course, Borland software. They used Turbo Pascal to design the boat's hull. They used Reflex: The Analyst® to maintain their databases and to display plots while the boat was sailing. And when it came time to do some mathematical modeling, again they turned to Borland for its inimitable software and chose the Toolbox.

Simply speaking, the problem they had was one of “precision monitoring.” It takes a crew of very highly skilled sailors to compete in America’s Cup races, but even the best skippers cannot act with sufficient precision to win. A typical race lasts for several hours, and the winner usually wins by only a few feet.

The electronic equipment on a boat can sense with reasonable accuracy all of the crucial variables: boat velocity, wind velocity, boat direction, boat position, and so on. This data must then be made available to the skipper in a coherent form, and he/she must decide at what angle to place the rudder based on that information. The problem is too complex to rely on intuition alone.

Even just displaying the velocity is more complex than you might think at first. When sailing on the ocean, the waves are big enough that the velocity is in constant flux. Fortunately, the fluctuations due to the waves represents a steadily periodic force. By using the *Fourier transforms* in Chapter 10 of the Toolbox, the crew was able to identify the periodic portion of the velocity and subtract it out. The result: the velocity as a function of time but with the wave fluctuations eliminated. The graph of this modified velocity is much smoother, and allows the skipper to tell much more quickly and accurately whether the boat is accelerating or decelerating.

To measure the acceleration quantitatively, the crew used the fact that the acceleration is the derivative of the velocity. They were able to do this easily with the *differentiation* routines in Chapter 4 of the Toolbox. They were also able to directly measure the distance travelled by using the *integration* routines in Chapter 5, and the fact that distance is the integral of the speed.

Perhaps the most difficult problem in navigating a sailboat is aiming the rudder. You can’t just aim the boat in the direction that you want to go, rather you have to pick a direction that you can sail rapidly, depending on the wind direction. An experienced skipper can judge this pretty well, but not well enough. Every boat is a little different, and the best way to handle one boat is not necessarily the best way to handle another.

So, the team ran extensive trial races with the boat to gather data on how the boat performed in various circumstances. The data was collected automatically by electronic instruments on board, and stored digitally on floppy disks. They then used Reflex to manage the data and to display graphs. But they lacked the tools to relate their data to the data they would have under actual racing conditions.

In order to predict the behavior of their boat in an actual race, the team created a model from their collected data using the *least-squares* routines in Chapter 9 of the Toolbox. With the least-squares routines, you can create a multiparameter model and then find the values of the parameters that make the model most accurately fit the data. With a mathematical model of the boat’s behavior, the team was then able to predict how the boat would perform under circumstances similar but not identical to its practices.

This, of course, is just one of many applications of this Toolbox. Now, let's go on to the fundamentals.

The Distribution Disks

All of the Toolbox routines are contained on three disks. (Note, however, that MS-DOS® users will receive two disks; Disk 3 is for PC-DOS users only.) Each disk has packed files corresponding to chapters in the manual. Use the program UNPACK.EXE to unpack the files, described in the next section, "Installation."

The files for each chapter are self-contained and do not require any files from any other chapter, with these exceptions:

- All files require Turbo Pascal (not included).
- Most files require COMMON.INC, located on Disk 1.
- The files for Chapter 11 require files from Chapters 9 and 10, as well as the Turbo Pascal Graphics Toolbox (not included).

The numerical analysis routines are in the files with the .INC suffix. The files with the .PAS suffix are demonstration programs. To run a demonstration program, get into Turbo Pascal and load the .PAS file of your choice. The menus are self-explanatory. The .DAT files contain input data for specific .PAS files.

If you're a PC-DOS user with an IBM color graphics monitor or compatible, you can run LSQIBM.COM or FFTIBM.COM from Disk 3 to see a quick graphic demonstration of the power and usefulness of the Toolbox. These routines require the files SAMP11A.DAT, SAMP11B.DAT, 4X6.FON, 8X8.FON, and ERROR.MSG to be on the current directory. (These files are also on Disk 3.)

Contents of the enclosed disks:

Disk 1

README

README.COM (program to display README file)

UNPACK.EXE (installation program to unpack chapters)

COMMON.INC (used throughout the Toolbox)

COMMON2.INC (for Turbo Pascal 2.0 users)

CHAP2 (packed file with routines for Chapter 2)

CHAP3 (packed file with routines for Chapter 3)

CHAP4 (packed file with routines for Chapter 4)

CHAP5 (packed file with routines for Chapter 5)

CHAP6 (packed file with routines for Chapter 6)

CHAP7 (packed file with routines for Chapter 7)

Disk 2

UNPACK.EXE (installation program to unpack chapters)
CHAP8 (packed file with routines for Chapter 8)
CHAP9 (packed file with routines for Chapter 9)
CHAP10 (packed file with routines for Chapter 10)
CHAP11 (packed file with routines for Chapter 11; PC-DOS users only)

Disk 3 (PC-DOS users only)

README
README.COM (program to display README file)
LSQIBM.COM (requires IBM graphics monitor)
FFTIBM.COM (requires IBM graphics monitor)
LSQHERC.COM (requires Hercules graphics card)
FFTHERC.COM (requires Hercules graphics card)
SAMP11A.DAT (data file for LSQ*.COM)
SAMP11B.DAT (data file for FFT*.COM)
14X9.FON (from the Graphix Toolbox)
4X6.FON (from the Graphix Toolbox)
8X8.FON (from the Graphix Toolbox)
ERROR.MSG (from the Graphix Toolbox)

Installation

The files CHAP2 through CHAP11 are packed files corresponding to the chapters in this manual. In order to use these files, you must first unpack them with UNPACK.EXE. The syntax is as follows:

```
UNPACK packed-file-name target-drive
```

For example, the files for Chapter 2 can be extracted and put on the current directory on drive C by placing Disk 1 in drive A, changing the logged drive to A by typing A: at the DOS prompt, and then typing

```
UNPACK CHAP2 C:
```

Wildcards are okay to use for the packed file name, and a directory can be specified with the target drive if it ends with a backslash (\). For example, all of the packed files on disk can be placed in a directory C:\NUMERIC by doing the following:

```
UNPACK CHAP* C:\NUMERIC\
```

if the directory C:\NUMERIC already exists.

You may wish to copy the packed files onto your hard disk, and then unpack them as you need them.

Note: These files are not copy protected. All files are ordinary DOS files; there are no hidden files. The unpacking program only extracts ordinary text files—it will not create directories, modify the distribution disk, create hidden or protected files, or do anything unexpected.

Contents of the packed files:

CHAP2 (packed file) “Roots to Equation in One Variable”

BISECT.INC	NEWTDEFL.PAS
BISECT.PAS	RAPHSON.INC
LAGUERRE.INC	RAPHSON.PAS
LAGUERRE.PAS	RAPHSON2.PAS
MULLER.INC	SECANT.INC
MULLER.PAS	SECANT.PAS
NEWTDEFL.INC	

CHAP3 (packed file) “Interpolation”

CUBE_CLA.INC	SAMPLE3B.DAT
CUBE_CLA.PAS	SAMPLE3C.DAT
CUBE_FRE.INC	SAMPLE3D.DAT
CUBE_FRE.PAS	SAMPLE3E.DAT
DIVDIF.INC	SAMPLE3F.DAT
DIVDIF.PAS	SAMPLE3G.DAT
LAGRANGE.INC	SAMPLE3H.DAT
LAGRANGE.PAS	SAMPLE3I.DAT
SAMPLE3A.DAT	

CHAP4 (packed file) “Numerical Differentiation”

DERIV.INC	DERIV2FN.INC
DERIV.PAS	DERIV2FN.PAS
DERIV2.INC	INTERDRV.INC
DERIV2.PAS	INTERDRV.PAS
DERIVFN.INC	SAMPLE4A.DAT
DERIVFN.PAS	SAMPLE4B.DAT

CHAP5 (packed file) “Numerical Integration”

ADAPGAUS.INC	ROMBERG.PAS
ADAPGAUS.PAS	SIMPSON.INC
ADAPSIMP.INC	SIMPSON.PAS
ADAPSIMP.PAS	TRAPZOID.INC
ROMBERG.INC	TRAPZOID.PAS

CHAP6 (packed file) “Matrix Routines”

DET.INC	INVERSE.INC
DET.PAS	INVERSE.PAS
DIRFACT.INC	PARTPIVT.INC
DIRFACT.PAS	PARTPIVT.PAS
GAUSELIM.INC	SAMPLE6A.DAT
GAUSELIM.PAS	SAMPLE6B.DAT
GAUSSIDL.INC	SAMPLE6C.DAT
GAUSSIDL.PAS	SAMPLE6D.DAT

CHAP7 (packed file) “Eigenvalues and Eigenvectors”

INVPOWER.INC	POWER.PAS
INVPOWER.PAS	SAMPLE7A.DAT
JACOBI.INC	WIELANDT.INC
JACOBI.PAS	WIELANDT.PAS
POWER.INC	

CHAP8 (packed file) “Initial Value and Boundary Value Methods”

ADAMS_1.INC	RUNGE_2.PAS
ADAMS_1.PAS	RUNGE_N.INC
LINSHOT2.INC	RUNGE_N.PAS
LINSHOT2.PAS	RUNGE_S1.INC
RKF_1.INC	RUNGE_S1.PAS
RKF_1.PAS	RUNGE_S2.INC
RUNGE_1.INC	SHOOT2.INC
RUNGE_1.PAS	SHOOT2.PAS
RUNGE_2.INC	

CHAP9 (packed file) “Least-Squares Approximations”

EXP.LSQ	POLY.LSQ
FOURIER.LSQ	POWER.LSQ
LEAST.INC	SAMPLE9A.DAT
LEAST.PAS	USER.LSQ
LOG.LSQ	

CHAP10 (packed file) “Fast Fourier Transform Routines”

COMPCNVL.INC	FFTPROGS.PAS
COMPCORR.INC	REALCNVL.INC
COMPFFT.INC	REALCORR.INC
FFT87B2.INC	REALFFT.INC
FFT87B4.INC	SAMP10A.DAT
FFTB2.INC	SAMP10B.DAT
FFTB4.INC	SAMP10C.DAT

CHAP11 (packed file) “Graphics Programs”/PC-DOS only

FFTDemo.PAS	LEAST.MOD
GENERIC.LSQ	LSQDEMO.PAS
GRAPHIX.EGA	
GRAPHIX.HGC	
IOCHECK.INC	

All sample programs call the include file `COMMON.INC` from the disk. This file includes procedures that are common to all sample programs. When copying any of the sample programs to a disk, be sure to also copy the file `COMMON.INC` to that disk or the sample programs will not compile.

To use the sample programs with Turbo Pascal version 2.0, rename `COMMON2.INC` (which is on Disk 1) to `COMMON.INC`. (You may wish to preserve a copy of the original `COMMON.INC` file by first copying it to a file called `COMMON3.INC`). If you run the sample programs with version 2.0 and do not make this change, the programs will compile but will handle I/O errors incorrectly.

We have made the sample programs general and easy to use. For example, numerical input can originate from the keyboard (where improper input is trapped) or from a text file; output can be sent to the printer, screen, or text file; other refinements are also included. Since, to a beginner, the supporting code may obscure the simplicity of calling the procedure, we have included a minimal sample program for *Newton-Raphson's* method of root-finding (`RAPHSON2.INC`).

The Graphics Demos

Because graphic displays are often an essential part of numerical analysis, we have included two demonstration programs (for PC-DOS users only) that involve display numerical results. As previously stated, graphics hardware is not necessary for this Toolbox, but it is required for these two graphics programs. The programs are built with subsets of the Turbo Pascal Graphix Toolbox; there are separate versions for systems with the Hercules Monochrome Graphics card and the IBM Color Graphics card (or good emulations of these cards).

The demonstration programs are on Disk 3. For instructions about how to run or recompile them, see Chapter 11.

Data Types and Defined Constants

Data types that might be confused with those in the calling program have been prefixed with the letters TN (for Turbo Numerical); for example, *TNmatrix* or *TNvector*. You must define these variable types in your top-level program for two reasons. First, you will probably need to use this type in your top-level program, and the type must be defined to have the same scope as the Toolbox procedure. Second, you will want to dimension arrays based on your particular needs. For example, the *Lagrange* procedure requires the definition

```
type TNvector = array[0..TNArraySize] of Real;
```

The identifier *TNArraySize* is never referred to in any of the include files. It should be optimized by the user, although we have set a default value in each of the sample programs. It may be replaced with an integer or byte.

TNNearlyZero is the only defined constant that must be changed when standard Turbo Pascal is used (instead of Turbo-87); it should be changed to $1\text{E} - 7$. (Without changing this constant, you will get a syntax error when compiling with standard Turbo Pascal.) For Turbo-87, the default value of this constant is $1\text{E} - 15$. (There are a few exceptions to these default values; where appropriate, the “Comments” section of each routine indicates the exceptions.)

Compiler Directives

Aside from the usual default values of the compiler directives in standard Turbo Pascal, we have set the compiler directive to `{$R+}` in all include files that use arrays, and to `{$I-}` in all sample programs. The first directive checks to see that all array-indexing operations are within the defined bounds and all assignments to scalar and subrange variables are within range. The latter directive disables I/O error-checking. All the sample programs have their own I/O error-checking procedures (loaded in the file `COMMON.INC`), so that the `{$I-}` directive must remain disabled in the sample programs. The array checker `{$R+}` should always be active, since the performance penalty is slight and the advantages are significant.

Roots to Equations in One Variable

The routines in this chapter are for finding the roots of a single equation in one real variable. A typical problem is to solve

$$x * \exp(x) - 10 = 0$$

In general, the routines find a value of x , where x is a scalar real variable, satisfying

$$f(x) = 0.0$$

where f is a real-valued function that you program in Pascal.

All of the methods are *approximate* methods, meaning that they find an approximate value of x that makes $f(x)$ close to zero. Because of round-off error, it is usually not possible to find the exact value of x . Furthermore, they are all *iterative* methods, meaning that you specify some initial guess that is some value for x , which you think is reasonably close to the solution. The routine repeats some calculations that replace the guess x with a more accurate guess until the required level of accuracy is achieved.

The *bisection* method (BISECT.INC) returns an approximation to a root of a real continuous function of the real variable x . This method always converges (as long as the function changes signs at a root), but may do so relatively slowly.

The *Newton-Raphson* method (RAPHSON.INC) also returns an approximation to a root of a real function f of the real variable x . When this algorithm converges, it is usually faster than the bisection method. If more than one root of a polynomial equation is desired, then use *Newton-Horner's* method (NEWTDEFL.INC).

The *secant* method (SECANT.INC) is similar to the Newton-Raphson method, but doesn't require knowledge of the first derivative of the function. Consequently, it is more flexible than the Newton-Raphson method, though somewhat slower.

Newton-Horner's method (NEWTDEFL.INC) applies Newton's method to real polynomials. It also uses *deflation* techniques to attempt to approximate all the real roots of a real polynomial. Both the Newton-Horner and Newton-Raphson methods are faster than the bisection and secant methods, but are undefined if $|f'(x)| \leq TN\text{NearlyZero}$. This is less of a problem on machines with a high-precision math coprocessor, since $TN\text{NearlyZero}$ is smaller.

The Newton-Horner and Newton-Raphson methods both converge around multiple roots, although convergence is slow. These algorithms depend upon an initial approximation of the root. If the initial approximation is not sufficiently close to the root, the Newton methods may not converge. In some instances, an initial choice may lead to successive iterations that oscillate indefinitely about a value of x usually associated with a relative minimum or relative maximum of f . In either case, the bisection method could be used to determine the root or to determine a close approximation to the root that can be employed as an initial approximation in the Newton-Raphson or Newton-Horner methods.

Müller's method (MULLER.INC) returns an approximation to a root (possibly complex) of a complex function of the complex variable x . Although Müller's method can approximate the roots of polynomials, we recommend that you use Newton-Horner's method, the secant method, or (in the case of complex polynomials) *Laguerre's* method to find the roots of polynomials.

Laguerre's method (LAGUERRE.INC) attempts to approximate all the real and complex roots of a real or complex polynomial. Laguerre's method is very reliable and quick, even when converging to a multiple root. This is the best general method to use with polynomials.

A caution when solving polynomial equations: Polynomials can be ill-conditioned, in the sense that small changes in the coefficients may lead to large changes in the roots.

Stopping Criteria

All the root-finding routines use the function *TestForRoot* to determine if a root has been found.

```
function TestForRoot(X, OldX, Y, Tol : Real) : Boolean;

(*****)
(* Here are four stopping criteria. If you wish to *)
(* change the active criteria, simply comment off the current *)
(* criteria (including the appropriate or) and remove the comment *)
(* brackets from the criteria (including the appropriate or) you *)
(* wish to be active. *)
(*****)

begin
  TestForRoot :=
    (ABS(Y) <= TNNearlyZero)
    or
    (ABS(X - OldX) < ABS(OldX*Tol))
    or
    (ABS(X - OldX) < Tol)
    or
    (ABS(Y) <= Tol)
  (*****)
end; { procedure TestForRoot }
```

The four separate tests provided by function *TestForRoot* may be used in any combination. The default criteria tests the absolute value of *Y* and the relative change in *X*. If you wish to change the active criteria, simply comment off the current criteria (including the appropriate *or*) and remove the comment brackets from the criteria (including the appropriate *or*) you wish to be active.

The first criterion simply checks to see if *Y* is zero (*TNNearlyZero* is defined at the beginning of the procedure). This criterion should usually be kept active.

The second criterion examines the relative change in *X* between iterations. To avoid division by zero errors, *OldX* has been multiplied through the inequality.

The third criterion checks the absolute change in *X* between iterations.

The fourth criterion determines the absolute difference between *Y* and the allowable tolerance. **Note:** The parameter *Tol*(erance) means something different in each test. Be sure you know which tests are active when you input a value for *Tol*.

Root of a Function Using the Bisection Method (BISECT.INC)

Description

This method (Burden and Faires 1985, 28 ff.) provides a procedure for finding a root of a real continuous function f , specified by the user on a user-supplied real interval $[a,b]$. The functions $f(a)$ and $f(b)$ must be of opposite signs. The algorithm successively bisects the interval and converges to the root of the function. You must also specify the desired accuracy to which the root should be approximated.

User-Defined Function

function TNTargetF(x : Real) : Real;

The procedure *Bisect* determines the roots of this function.

Input Parameters

LeftEnd:Real; Left end of the interval

RightEnd:Real; Right end of the interval

Tol:Real; Indicates accuracy of solution

MaxIter:Real; Maximum number of iterations permitted

The preceding parameters must satisfy the following conditions:

1. $LeftEnd < RightEnd$.
2. $TNTargetF(LeftEnd) * TNTargetF(RightEnd) < 0$; the endpoints must have opposite signs.
3. $Tol > 0$.
4. $MaxIter \geq 0$.

Output Parameters

Answer:Real; An approximate root of $TNTargetF$
fAnswer:Real; The value of the function at the value *Answer*
Iter:Integer; Number of iterations to find answer
Error:Byte; 0: No error
 1: $Iter > MaxIter$
 2: Endpoints are of the same sign
 3: $LeftEnd > RightEnd$
 4: $Tol \leq 0$
 5: $MaxIter < 0$

If $Error = 1$ (maximum number of iterations exceeded), *Answer* is set to the last x value tested and *fAnswer* is set to $TNTargetF(Answer)$. If $Error > 1$, then the other output parameters are not defined.

Syntax of the Procedure Call

```
Bisect(LeftEnd, RightEnd, Tol, MaxIter, Answer, fAnswer, Iter, Error);
```

The procedure *Bisect* determines the roots of function $TNTargetF$.

Comments

If a root occurs at a relative maximum or relative minimum, the bisection method will be unable to locate that value of p if p does not occur as an endpoint of a subinterval.

Convergence is determined with the Boolean function *TestForRoot* described at the beginning of this chapter.

Sample Program

The sample program BISECT.PAS provides I/O functions that demonstrate the bisection algorithm. To modify this program for your own function, simply change the definition of function $TNTargetF$. Note that the file BISECT.INC is included after the function $TNTargetF$ is defined.

Example

Problem. Determine the solution to the equation $\cos(x) = x$.

1. Write the following code for function *TNTargetF* into BISECT.PAS:

```
{----- HERE IS THE FUNCTION -----}  
  
function TNTargetF(x : Real) : Real;  
begin  
    TNTargetF := Cos(x) - x;  
end;  
    { function TNTargetF }  
  
{-----}
```

2. Run BISECT.PAS:

Left endpoint: 0
Right endpoint: 100

Tolerance (> 0, default = 1.000E-08): 1E-6

Maximum number of iterations (>= 0, default = 100)? 100

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

Left endpoint: 0.0000000000000E+000
Right endpoint: 1.0000000000000E+002
Tolerance: 1.0000000000000E-006
Maximum number of iterations: 100

Number of iterations: 28
Calculated root: 7.39085301756859E-001
Value of the function
at the calculated root: -2.82073423951701E-007

Root of a Function Using the Newton-Raphson Method (RAPHSON.INC)

Description

This example uses Newton-Raphson's algorithm (Burden and Faires 1985, 42 ff.) to find a root of a real user-specified function when the derivative of the function and an initial guess are given. The algorithm constructs the tangent line at each iterate approximation of the root. The intersection of the tangent line with the x -axis provides the next iterate value of the root. You must specify the desired tolerance to which the root should be approximated.

User-Defined Functions

```
function TNTargetF(x : Real) : Real;
```

```
function TNDerivF(x : Real) : Real;
```

The procedure *Newton Raphson* determines the roots of the function *TNTargetF*.

The function *TNDerivF* must be the first derivative of function *TNTargetF*.

Input Parameters

Guess:Real; User's initial approximation to the root

Tol:Real; Tolerance in answer (see "Comments")

MaxIter:Integer; Maximum number of iterations permitted

The preceding parameters must satisfy the following conditions:

1. *Tol* > 0
2. *MaxIter* > 0

Output Parameters

Root:Real; Approximate root.
Value:Real; Value of the function at the approximate root.
Deriv:Real; Value of the derivative at the approximated root.
Iter:Integer; Number of iterations needed to find the root.
Error:Byte; 0: No error.
 1: $Iter < MaxIter$.
 2: The slope is zero (see “Comments”).
 3: $Tol \leq 0$.
 4: $MaxIter < 0$.

If a root is found, it is returned along with the value of the function at the root (which, of course, should be close to zero) and the value of the derivative at the root. If $Error \leq 2$, the data from the last iteration is returned.

Syntax of the Procedure Call

Newton_Raphson(Guess, Tol, MaxIter, Root, Value, Deriv, Iter, Error);

Comments

Newton’s method involves division by the value of the derivative of the function. Should the algorithm attempt to do any calculations at a point where the derivative is less than $TNNearlyZero$, the routine will stop and return an error message ($Error = 2$).

Convergence is determined with the Boolean function *TestForRoot* described at the beginning of this chapter.

Sample Program

The sample program RAPHSON.PAS provides I/O functions that demonstrate the Newton-Raphson algorithm. Note that the file RAPHSON.INC is included after the functions *TNTargetF* and *TNDerivF* are defined.

The program RAPHSON2.PAS also provides I/O functions that demonstrate the Newton-Raphson method. It is an extremely bare-bones program and is provided

for the newcomer to Turbo Pascal who wants to see a simple, straightforward application of a Toolbox routine.

Example

Problem. Determine the solution to the equation $\cos(x) = x$.

1. Code the following two functions into RAPHSON.PAS (or RAPHSON2.PAS):

```
{----- HERE IS THE FUNCTION -----}
```

```
function TNTargetF(x : Real) : Real;  
begin  
    TNTargetF := Cos(x) - x;  
end;           { function TNTargetF }
```

```
{-----}
```

```
{----- HERE IS THE DERIVATIVE -----}
```

```
function TNDerivF(x : Real) : Real;  
begin  
    TNDerivF := -Sin(x) - 1;  
end;           { function TNDerivF }
```

```
{-----}
```

2. Run RAPHSON.PAS:

Initial approximation to the root: 0

Tolerance (> 0, default = 1.000E-08): 1E-6

Maximum number of iterations (>= 0, default = 100): 100

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

Initial approximation: 0.0000000000000E+000
Tolerance: 1.0000000000000E-006
Maximum number of iterations: 100

Number of iterations: 5
Calculated root: 7.39085133215161E-001
Value of the function
at the calculated root: 0.0000000000000E+000
Value of the derivative
of the function at the
calculated root: -1.67361202918321E+000

Here is the RAPHSON2.PAS version of the same function:

Initial approximation to the root: 0
Tolerance: 1E-6
Maximum number of iterations: 100

Error = 0

Number of iterations: 5
Root: 7.39085133215161E-001
Value of the
function at the root: 0.00000000000000E+000
Derivative of the
function at the root: -1.67361202918321E+000

Root of a Function Using the Secant Method (SECANT.INC)

Description

This example uses the secant method (Gerald and Wheatley 1984, 11-13) to find a root of a user-specified real function given two initial real approximations to the root. The secant method constructs a secant through the two points specified by the initial approximations. The intersection of this line and the x -axis is used as the next best approximation to the root. The approximation to the root and its predecessor are used to construct the next secant line. The process continues until a root is approximated with specified accuracy or until a specified number of iterations have been exceeded.

User-Defined Function

function TNTargetF(x : Real) : Real;

The procedure *Secant* will determine the roots of this function.

Input Parameters

Guess1:Real; User's first approximation to the root
Guess2:Real; User's second approximation to the root
Tol:Real; Indicates accuracy in solution
MaxIter:Integer; Maximum number of iterations permitted

The preceding parameters must satisfy the following conditions:

1. $Tol > 0$
2. $MaxIter \geq 0$

Output Parameters

Root:Real; Approximate root.
Value:Real; Value of the function at the approximate root.
Iter:Integer; Number of iterations needed to find the root.
Error:Byte; 0: No error.
 1: $Iter > MaxIter$.
 2: The slope is zero (see “Comments”).
 3: $Tol \leq 0$.
 4: $MaxIter < 0$.

If a root is found, it is returned with the value of the function at the root (which, of course, should be nearly zero). If $Error \leq 2$, then the data from the last iteration is returned.

Syntax of the Procedure Call

Secant(Guess1, Guess2, Tol, MaxIter, Root, Value, Iter, Error);

The procedure *Secant* determines the roots of the function *TNTargetF*.

Comments

The secant algorithm constructs a line through two points and finds the intersection of that line with the x -axis. If the line has a slope whose absolute values are less than *TNNearlyZero* (that is, the two points have the same y -value), then it has no intersection with the x -axis (or infinitely many if it lies on the x -axis) and the algorithm will no longer continue. If this happens, Error 2 is returned. Error 2 will also be returned if the absolute difference of the two initial approximations (*Guess1* and *Guess2*) is less than *TNNearlyZero*.

Convergence is determined with the Boolean function *TestForRoot* described at the beginning of this chapter.

Sample Program

The sample program SECANT.PAS provides I/O functions that demonstrate the secant algorithm. Note that the file SECANT.INC is included after the function *TNTargetF* is defined.

Example

Problem. Determine the solution to the equation $\cos(x) = x$.

1. Write the following code for procedure *TNTargetF* into SECANT.PAS:

```
{----- HERE IS THE FUNCTION -----}  
  
function TNTargetF(x : Real) : Real;  
begin  
    TNTargetF := Cos(x) - x;  
end;  
    { function TNTargetF }  
  
{-----}
```

2. Run SECANT.PAS:

First initial approximation to the root: 0

Second initial approximation to the root: 1

Tolerance (> 0, default = 1.000E-08): 1E-8

Maximum number of iterations (>= 0, default = 100): 100

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

First initial approximation: 0.0000000000000E+000

Second initial approximation: 1.0000000000000E+000

Tolerance: 1.0000000000000E-008

Maximum number of iterations: 100

Number of iterations: 6

Calculated root: 7.39085133215161E-001

Value of the function

at the calculated root: 0.0000000000000E+000

Real Roots of a Real Polynomial Equation Using the Newton-Horner Method with Deflation (NEWTDEFL.INC)

Description

This example uses Newton-Horner's algorithm and deflation (see RAPHSON.INC in this chapter for a description of Newton's method). Newton-Horner is the Newton-Raphson method applied to polynomials (Burden and Faires 1985, 42 ff). *Deflation* is used to find several roots of a user-specified real polynomial given an initial guess specified by the user. This procedure approximates a real root and then removes the corresponding linear factor from the given polynomial. The newly obtained (deflated) polynomial is then analyzed for a real root. This process continues until a quadratic remains, the remaining roots are complex, or the algorithm is unable to approximate the remaining real roots. Should the polynomial contain two complex roots, they may be determined using the quadratic formula. You must specify (at most) the tolerance to which the roots should be approximated.

User-Defined Types

```
TNvector = array[0..TNArraySize] of Real;  
TNIntVector = array[0..TNArraySize] of Integer;
```

Input Parameters

InitDegree:Integer;	Degree of user-defined polynomial
InitPoly:TNvector;	Coefficients of user-defined polynomial
Guess:Real;	User's initial approximation
Tol:Real;	Indicates accuracy in solution
MaxIter:Integer;	Maximum number of iterations permitted

The preceding parameters must satisfy the following conditions:

1. $InitDegree > 0$
2. $Tol > 0$
3. $MaxIter \geq 0$
4. $InitDegree \leq TNArrarySize$

TNArrarySize fixes an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector*. *TNArrarySize* is *not* a variable name and is never referenced by the procedure; hence there is no test for condition 4. If condition 4 is violated, the program will crash with an Index Out of Range error (assuming the directive `{R+}` is active).

Output Parameters

Degree:Integer;	Degree of the deflated polynomial (> 2 if some of the roots are not approximated).
NumRoots:Integer;	Number of roots found.
Poly:TNvector;	Coefficients of the deflated polynomial.
Root:TNvector;	Real part of all roots found.
Imag:TNvector;	Imaginary part of all roots found (nonzero for 2 at most).
Value:TNvector;	Value of the polynomial at each approximate root.
Deriv:TNvector;	Value of the derivative at each found root.
Iter:TNIntVector;	Number of iterations required to find each root.
Error:Byte;	0: No error. 1: Maximum number of iterations exceeded. 2: The slope is zero (see "Comments"). 3: $Degree \leq 0$. 4: $Tol \leq 0$. 5: $MaxIter < 0$.

If a root is found, it is returned with the value of the polynomial at that root (which should be close to zero) and with the value of the derivative at that root. If the last two roots are complex (only two can be complex, since they are evaluated by the quadratic formula), then the value and derivative at those points are arbitrarily set to zero. If all the roots have not been found, then the unsolved deflated polynomial is also returned.

Syntax of the Procedure Call

```
Newton_Horn_Defl(InitDegree, InitPoly, Guess, Tol, MaxIter, Degree,  
                NumRoots, Poly, Root, Imag, Value, Deriv, Iter, Error);
```

Comments

Newton's method involves division by the derivative of the function. Should the algorithm attempt to do any calculations at a point where the absolute values of the derivative are less than *TNNearlyZero*, the routine stops and returns an error message (Error = 2).

Convergence is determined with the Boolean function *TestForRoot* described at the beginning of this chapter.

Sample Program

The sample program NEWTDEFL.PAS provides I/O functions that demonstrate the Newton-deflation algorithm.

Input Files

It is possible to input the coefficients from a text file. The format for the text file is as follows:

1. The degree of the polynomial
2. The coefficients in descending order, beginning with the leading coefficient and decreasing to the constant term

Spaces or carriage returns can be used to separate the data. It does not matter whether the file ends with a carriage return; for example, the polynomial

$$F(x) = x^3 - 2x$$

could be entered in a text file as

3 1 0 -2 0

Example

Problem. Determine the roots to the 7th degree polynomial:

$$x^6 + x^5 - 49x^4 + 69x^3 + 120x^2 + 98x - 240$$

Run NEWTDEFL.PAS:

(K)eyboard or (F)ile input of data? K

Degree of the polynomial (≤ 30)? 6

Input the coefficients of the polynomial
where Poly[n] is the coefficient of x^n

Poly[6] = 1
Poly[5] = 1
Poly[4] = -49
Poly[3] = 69
Poly[2] = 120
Poly[1] = 98
Poly[0] = -240

Initial approximation to the root: 0

Tolerance (> 0 , default = $1.000\text{E-}08$): $1\text{E-}8$

Maximum number of iterations (≥ 0 , default = 100): 100

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

Initial Polynomial:

Poly[6]: $1.000000000000000\text{E}+000$
Poly[5]: $1.000000000000000\text{E}+000$
Poly[4]: $-4.900000000000000\text{E}+001$
Poly[3]: $6.900000000000000\text{E}+001$
Poly[2]: $1.200000000000000\text{E}+002$
Poly[1]: $9.800000000000000\text{E}+001$
Poly[0]: $-2.400000000000000\text{E}+002$

Initial approximation: $0.000000000000000\text{E}+000$

Tolerance: $1.000000000000000\text{E-}008$

Maximum number of iterations: 100

Number of calculated roots: 6

Root 1

Number of iterations: 7

Calculated root: $3.000000000000000\text{E}+000$

Value of the function
at the calculated root: $4.83169060316868\text{E-}013$

Value of the derivative
of the function at
the calculated root: $-7.47999999999999\text{E}+002$

Root 2
 Number of iterations: 7
 Calculated root: 1.00000000000000E+000
 Value of the function
 at the calculated root: 0.00000000000000E+000
 Value of the derivative
 of the function at
 the calculated root: 3.60000000000000E+002

Root 3
 Number of iterations: 32
 Calculated root: -8.00000000000000E+000
 Value of the function
 at the calculated root: 0.00000000000000E+000
 Value of the derivative
 of the function at
 the calculated root: -6.43500000000000E+004

Root 4
 Number of iterations: 25
 Calculated root: 5.00000000000000E+000
 Value of the function
 at the calculated root: 0.00000000000000E+000
 Value of the derivative
 of the function at
 the calculated root: 3.84800000000000E+003

Root 5
 Number of iterations: 0
 Calculated root: -1.00000000000000E+000 + -1.00000000000000E+000i
 Value of the function
 at the calculated root: 0.00000000000000E+000
 Value of the derivative
 of the function at
 the calculated root: 0.00000000000000E+000

Root 6
 Number of iterations: 0
 Calculated root: -1.00000000000000E+000 + 1.00000000000000E+000i
 Value of the function
 at the calculated root: 0.00000000000000E+000
 Value of the derivative
 of the function at
 the calculated root: 0.00000000000000E+000

Complex Roots of a Complex Function Using Müller's Method (MULLER.INC)

Description

This example uses Müller's method (Burden and Faires 1985, 71–75) to find a possibly complex root of a user-defined complex function. The algorithm finds a root of a parabola defined by three distinct points of the given function. This approximation to the root and its two predecessors are used to construct the next parabola. This is repeated until the convergence criteria is satisfied. Müller's method has the advantage of nearly always converging; however, it is slow because it uses complex arithmetic. You must create a complex function, input an initial guess (which need not be very accurate), the tolerance in the answer, and the maximum number of iterations.

User-Defined Types

```
TNcomplex = record
    Re, Im:Real;
end;
```

User-Defined Procedure

```
procedure TNtargetF(x:TNcomplex; var y:TNcomplex);
```

The *Muller* procedure approximates a complex root of this function.

Input Parameters

Guess:TNcomplex; An initial guess
Tol:Real; Indicates accuracy in solution
MaxIter:Integer; Maximum number of iterations

The preceding parameters must satisfy the following conditions:

1. $Tol > 0$
2. $MaxIter \geq 0$

Output Parameters

Answer:TNcomplex; An approximate root of the function
yAnswer:TNcomplex; Value of the function at the approximate root
Iter:Integer; Number of iterations required to find the root
Error:Byte; 0: No error
 1: $Iter > MaxIter$
 2: Parabola could not be formed (see “Comments”)
 3: $Tol \leq 0$
 4: $MaxIter < 0$

If $Error \leq 2$, then the information from the last iteration is output.

Syntax of the Procedure Call

Muller(Guess, Tol, MaxIter, Answer, yAnswer, Iter, Error);

The procedure *Muller* approximates a complex root of function *TNTargetF*.

Comments

Müller’s method involves constructing a parabola from three points. If they all lie on a line whose slope in absolute value is less than *TNNearlyZero*, then a parabola that intersects the x -axis cannot be constructed. Such a construction will halt the algorithm and return $Error = 2$. Fortunately, this does not commonly occur.

Convergence is determined with the Boolean function *TestForRoot* described at the beginning of this chapter. Complex arithmetic is used.

Sample Program

The sample program MULLER.PAS provides I/O functions that demonstrate Müller's method.

The user-defined function is contained in the procedure *TNTargetF*. It is necessary to separately define the real and complex parts of the function. To define the complex function $F(x)$, you must code the following definitions:

```
y.Re := Re[F(x.Re + ix.Im)];  
y.Im := Im[F(x.Re + ix.Im)];
```

where i is the square root of -1 .

For example, the complex function $F(x) := \exp(x)$ would be coded like this:

```
y.Re := exp(x.Re) * cos(x.Im);  
y.Im := exp(x.Re) * sin(X.Im);
```

Note that the procedure *TNTargetF* is defined before MULLER.INC is included.

Example

Problem. Find a solution to the complex equation $\cos(x) = x$.

1. First, code the following procedure *TNTargetF* into MULLER.PAS:

```
(*----- HERE IS THE FUNCTION -----*)  
  
procedure TNTargetF(x : TNcomplex; var y : TNcomplex);  
  
begin { this is the complex function y = Cos(x) - x }  
  y.Re := Cos(x.Re)*(Exp(-x.Im) + Exp(x.Im))/2 - x.Re;  
  y.Im := Sin(x.Re)*(Exp(-x.Im) - Exp(x.Im))/2 - x.Im;  
end;  
      { procedure TNTargetF }  
  
(*-----*)
```

2. Run MULLER.PAS:

Initial approximation to the root:

Re(Approximation)= -4

Im(Approximation)= 4

Tolerance (> 0, default = 1.000E-08): 1E-6

Maximum number of iterations (>= 0, default = 100): 100

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

Initial approximation: $-4.000000000000000E+000 + 4.000000000000000E+000i$

Tolerance: $1.000000000000000E-006$

Maximum number of iterations: 100

Number of iterations: 18

Calculated root: $-9.10998745393294E+000 + 2.95017086170180E+000i$

Value of the function

at the calculated root: $-1.42534872793476E-011 + 3.75033337718378E-011i$

Complex Roots of a Complex Polynomial Using Laguerre's Method and Deflation (LAGUERRE.INC)

Description

This example uses Laguerre's method (Ralston and Rabinowitz 1978, 380–383) and linear deflation to find the possibly complex roots of a complex (or real) polynomial. You must input the coefficients of the polynomial, an initial guess, the tolerance with which to find the answer, and the maximum number of iterations.

User-Defined Types

```
TNcomplex = record
    Re, Im:Real;
end;

TNIntVector = array[0..TNArraySize] of Integer;

TNCompVector = array[0..TNArraySize] of TNcomplex;
```

Input Parameters

Degree:Integer;	Degree of the user's polynomial
Poly:TNvector;	Coefficients of the user's polynomial
InitGuess:TNcomplex;	Initial guess of the root
Tol:Real;	Indicates accuracy in solution
MaxIter:Integer;	Maximum number of iterations

The preceding parameters must satisfy the following conditions:

1. *degree* > 0
2. *Tol* > 0
3. *MaxIter* ≥ 0
4. *degree* ≤ *TNArraySize*

TNArraySize fixes an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector*. *TNArraySize* is not a variable name and is never referenced by the procedure; hence there is no test for condition 4. If condition 4 is violated, the program will crash with an Index Out of Range error (assuming the directive `{%R+}` is enabled).

Output Parameters

Degree:Integer;	Degree of the deflated polynomial
Poly:Integer;	Coefficients of deflated polynomial
NumRoots:Integer;	Number of approximate roots
Roots:TNCompVector;	Approximate roots
yRoots:TNCompVector;	Value of the polynomial at the approximate root
Iter:TNIIntVector;	Number of iterations required to find each root
Error:Byte;	0: No error 1: $Iter \geq MaxIter$ 2: $Degree \leq 0$ 3: $Tol \leq 0$ 4: $MaxIter < 0$

Syntax of the Procedure Call

```
Laguerre(Degree, Poly, InitGuess, Tol, MaxIter, NumRoots,
          Roots, yRoots, Iter, Error);
```

Comments

For some polynomials, certain starting values (*Guess*) will not yield convergence. If the routine does not converge to a solution, try a different starting value. Note that convergence is slower around multiple roots than around single roots.

Convergence is determined with the Boolean function *TestForRoot* described at the beginning of this chapter.

Sample Program

The sample program LAGUERRE.PAS provides I/O routines that demonstrate Laguerre's method.

Input Files

It is possible to input the coefficients from a text file. The format for the text file is as follows:

1. The degree of the polynomial
2. The real and imaginary parts of the coefficients in descending order, beginning with the leading coefficient and descending to the constant term

Spaces or carriage returns can be used to separate the data. It does not matter whether the file ends with a carriage return; for example, the polynomial

$$F(x) = x^4 - (2 + 2i)x^3 + 4ix^2 + (2 - 2i)x - 1$$

where i represents the square root of -1 , could be entered in a text file like this:

4 1 0 -2 -2 0 4 2 -2 -1 0

Example

Problem. Find all the roots to the complex polynomial

$$F(x) = x^4 - (2 + 2i)x^3 + 4ix^2 + (2 - 2i)x - 1$$

where i is the square root of -1 .

Run LAGUERRE.PAS:

(K)eyboard or (F)ile input of data? K

Degree of the polynomial (<= 30)? 4

Input the coefficients of the polynomial
where Poly[n] is the coefficients of x^n

Re(Poly[4]) = 1
Im(Poly[4]) = 0

Re(Poly[3]) = -2
Im(Poly[3]) = -2

Re(Poly[2]) = 0
Im(Poly[2]) = 4

Re(Poly[1]) = 2
Im(Poly[1]) = -2

Re(Poly[0]) = -1
Re(Poly[0]) = 0

Initial approximation:
Re(Approximation) = 0
Im(Approximation) = 0

Tolerance (> 0, default = 1.000E-08): 1E-6
Maximum number of iterations (>= 0, default = 100): 100

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

Initial Polynomial:

Poly[4]: 1.0000000000000E+000 + 0.0000000000000E+000i
Poly[3]: -2.0000000000000E+000 + -2.0000000000000E+000i
Poly[2]: 0.0000000000000E+000 + 4.0000000000000E+000i
Poly[1]: 2.0000000000000E+000 + -2.0000000000000E+000i
Poly[0]: -1.0000000000000E+000 + 0.0000000000000E+000i

Initial approximation: 0.0000000000000E+000 + 0.0000000000000E+000i
Tolerance: 1.0000000000000E-006

Maximum number of iterations: 100

Root 1

Number of iterations: 14

Calculated root: 9.99999949924438E-001 + 1.30979567319146E-008i

Value of the function at

the calculated root: -2.44249065417534E-015 + -4.67073566002583E-015i

Root 2

Number of iterations: 2

Calculated root: 1.00000001879739E+000 + -3.01914119849697E-009i

Value of the function at

the calculated root: -3.33066907387547E-016 + -1.03591657956252E-015i

Root 3

Number of iterations: 6

Calculated root: 2.54057206426756E-007 + 9.99999844722996E-001i

Value of the function at

the calculated root: -1.57873714101697E-013 + -8.07179393811825E-014i

Root 4

Number of iterations: 2

Calculated root: $-5.97412717946260E-008 + 1.00000003900209E+000i$

Value of the function at

the calculated root: $-9.32587340685131E-015 + -3.90669101062423E-015i$

The exact roots of this polynomial are

$$x = 1$$

$$x = 1$$

$$x = i$$

$$x = i$$

Interpolation

Interpolation is useful when some values of a function are known but others are required. For example, suppose values are known for a function $f(x)$ at $x = 2.3, 2.4, 2.5, 2.6, 2.7, 2.8$, and the value of $f(x)$ is desired at $x = 2.415$. The routines in this chapter provide the means to model to given values of $f(x)$ with an appropriate function, so that the function can be evaluated at other arbitrary points.

The goal of interpolation is to approximate the value of the function at a specified value of x , given N values of the function at N distinct points. This approximation will be a polynomial determined from the input data. The value of the polynomial at x will be returned as the approximation to the value of $f(x)$.

The *Lagrange* method (LAGRANGE.INC) accepts points in any order. The x -values need not be equally spaced. An interpolating polynomial is explicitly calculated. Although an interpolating polynomial can be useful for computing derivatives (and more), the Lagrange method is a lengthy process. Furthermore, high-degree polynomials may cause significant round-off error in some interpolations.

Newton's general divided-difference algorithm (DIVDIF.INC) does not require input to have equally spaced x -values, nor is it necessary that the x -values be in either ascending or descending order. For large amounts of data, the divided-difference routine (DIVDIF.INC) is more accurate than Lagrangian interpolation (LAGRANGE.INC).

If there are many input points, the Lagrange (LAGRANGE.INC) and the divided-difference (DIVDIF.INC) methods may result in high-degree polynomials whose oscillatory nature can produce an inaccurate approximation. This is espe-

cially true if the interpolation occurs at a value near the midpoint between adjacent input x -values. In such cases, the *cubic spline* methods (CUBE_FRE.INC and CUBE_CLA.INC) are preferable.

The cubic spline methods require that the x -values be entered in ascending order. The *clamped cubic spline* method (CUBE_CLA.INC) may yield more accurate results than the *free cubic spline* method (CUBE_FRE.INC), but requires knowledge of the first derivative of the function at the endpoints of the input data. When this information is not available, the free cubic spline routine should be used.

The values at which interpolation is to occur should lie in the closed interval bounded by the extreme values of the input x -values. The preceding methods will not give accurate approximations to values outside this interval (extrapolation).

Polynomial Interpolation Using Lagrange's Method (LAGRANGE.INC)

Description

This example provides an interpolation algorithm (Burden and Faires 1985, 84 ff; Horowitz and Sahni 1984, 429-430). Given a set of N data points (x,y) , the routine uses Lagrange polynomials to construct a polynomial to fit the data points. The degree of the polynomial is at most $N - 1$.

Note: The nature of high-degree polynomials may cause significant error if the algorithm is used with large amounts of data (about $N > 25$). In such cases, DIVDIF.INC, CUBE_FRE.INC, or CUBE_CLA.INC should be used. You must supply the data points and the x -values at which interpolation will take place.

User-Defined Types

```
TNvector = array[0..TNArraySize] of Real;  
TNmatrix = array[0..TNArraySize] of TNvector;
```

Input Parameters

The parameters for Lagrange:

NumPoints:Integer; Number of data points
XData:TNvector; The x -coordinates of the data points
YData:TNvector; The y -coordinates of the data points
NumInter:Integer; Number of interpolations
XInter:TNvector The x -coordinates at which interpolation is to take place

The preceding parameters must satisfy the following conditions:

1. The x -coordinates of the data points ($XInter$) must be unique.
2. $NumPoints, NumInter \leq TNArraySize$.
3. $NumPoints > 0$.

TNArraySize fixes an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector*. *TNArraySize* is *not* a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error (assuming the directive `{R+}` is active).

Output Parameters

YInter:TNvector; The interpolated values at *XInter*
Poly:TNvector; The coefficients of the interpolating polynomial
Error:Byte; 0: No error
 1: X-values of the data points not unique
 2: *NumPoints* < 1

Syntax of the Procedure Call

Lagrange(NumPoints, XData, YData, NumInter, XInter, YInter, Poly, Error);

Sample Program

The sample program LAGRANGE.PAS provides I/O functions that demonstrate the Lagrange interpolating algorithm.

Input Files

Data may be entered from a text file. The *x* and *y* coordinates should be separated by a space and followed by a carriage return. For example, data values of $\text{sqr}(x)$ could be entered in a text file as:

```
1 1
2 4
3 9
4 16
5 25
```


Example

Problem. Construct and use an interpolating polynomial for the cosine function between $x = 1$ degree and $x = 20$ degrees.

Run LAGRANGE.PAS:

(K)eyboard or (F)ile entry of the data points? F

File name? SAMPLE3A.DAT

(K)eyboard or (F)ile entry of the interpolated points? F

File name? SAMPLE3B.DAT

Direct output to one of the following:

- (S)creen
- (P)rinter
- (F)ile

The data:

1.0000000	9.99847695156391E-001
2.0000000	9.99390827019096E-001
3.0000000	9.98629534754574E-001
4.0000000	9.97564050259824E-001
5.0000000	9.96194698091746E-001
6.0000000	9.94521895368273E-001
7.0000000	9.92546151641322E-001
8.0000000	9.90268068741570E-001
9.0000000	9.87688340595138E-001
10.0000000	9.84807753012208E-001
11.0000000	9.81627183447664E-001
12.0000000	9.78147600733806E-001
13.0000000	9.74370064785235E-001
14.0000000	9.70295726275996E-001
15.0000000	9.65925826289068E-001
16.0000000	9.61261695938319E-001
17.0000000	9.56304755963035E-001
18.0000000	9.51056516295154E-001
19.0000000	9.45518575599317E-001
20.0000000	9.39692620785908E-001

The polynomial:

Poly[19]= -1.72986376643586E-028
Poly[18]= 3.57504241395844E-026
Poly[17]= -3.43926153199199E-024
Poly[16]= 2.04507188280402E-022
Poly[15]= -8.41710490427928E-021
Poly[14]= 2.54454663251946E-019
Poly[13]= -5.85115478257286E-018
Poly[12]= 1.04567701944119E-016
Poly[11]= -1.47131277721604E-015
Poly[10]= 1.64108936708876E-014
Poly[9]= -1.45382580196402E-013
Poly[8]= 1.02034999744286E-012
Poly[7]= -5.63354870368428E-012
Poly[6]= 2.41324946244329E-011
Poly[5]= -7.90989844502363E-011

```

Poly[4]= 4.05827439744465E-009
Poly[3]= -3.31023555675263E-010
Poly[2]= -1.52308331145220E-004
Poly[1]= -2.53687934078865E-010
Poly[0]= 1.00000000007368E+000

```

X	Interpolated Y value	Actual values
1.500	9.99657324975249E-001	9.99657324975557E-001
2.500	9.99048221581889E-001	9.99048221581858E-001
3.500	9.98134798421861E-001	9.98134798421867E-001
4.500	9.96917333733130E-001	9.96917333733128E-001
5.500	9.95396198367178E-001	9.95396198367179E-001
6.500	9.93571855676587E-001	9.93571855676587E-001
7.500	9.91444861373810E-001	9.91444861373810E-001
8.500	9.89015863361917E-001	9.89015863361917E-001
9.500	9.86285601537231E-001	9.86285601537231E-001
10.500	9.83254907563954E-001	9.83254907563955E-001
11.500	9.79924704620830E-001	9.79924704620830E-001
12.500	9.76296007119933E-001	9.76296007119933E-001
13.500	9.72369920397676E-001	9.72369920397677E-001
14.500	9.68147640378107E-001	9.68147640378108E-001
15.500	9.63630453208623E-001	9.63630453208623E-001
16.500	9.58819734868193E-001	9.58819734868193E-001
17.500	9.53716950748226E-001	9.53716950748227E-001
18.500	9.48323655206200E-001	9.48323655206199E-001
19.500	9.42641491092201E-001	9.42641491092178E-001
20.500	9.36672189247006E-001	9.36672189248398E-001

The data is taken from a function of which the derivative could be computed exactly. Though the actual values in the right-hand column are not displayed on screen, they are shown here to indicate the accuracy of the routine.

Interpolation Using Newton's Interpolary Divided-Difference Method (DIVDIF.INC)

Description

This example provides an interpolation algorithm. Given a set of data points (x,y) , the routine uses Newton's interpolary divided-difference equation to interpolate between the points (Burden and Faires 1985, 100–102). The data points must have unique x -values, but these values need not be evenly spaced nor set in any particular order. You must supply the data points and the x -values at which interpolation is to take place.

User-Defined Types

```
TNvector = array[0..TNArraySize] of Real;  
TNmatrix = array[0..TNArraySize] of TNvector;
```

Input Parameters

```
NumPoints:Integer;  Number of data points  
XData:TNvector;     The  $x$ -coordinates of the data points  
YData:TNvector;     The  $y$ -coordinates of the data points  
NumInter:Integer;   Number of interpolations  
XInter:TNvector     The  $x$ -coordinates at which interpolation is to take place
```

The preceding parameters must satisfy the following conditions:

1. The x -coordinates of the data points ($XInter$) must be unique.
2. $NumPoints, NumInter \leq TNArraySize$.
3. $NumPoints > 0$.

$TNArraySize$ fixes an upper bound on the number of elements in each vector. It is used in the **type** definition of $TNvector$. $TNArraySize$ is *not* a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error (assuming the directive $\{\$R+\}$ is active).

Output Parameters

YInter: TNvector; The interpolated values at *XInter*
Error: Byte; 0: No error
 1: X-values of the data points not unique
 2: *NumPoints* < 1

Syntax of the Procedure Call

`Divided_Difference(NumPoints, XData, YData, NumInter, XInter, YInter, Error);`

Sample Program

The sample program DIVDIF.PAS provides I/O functions that demonstrate Newton's interpolary divided-difference algorithm.

Input Files

Data may be entered from a text file. The *x* and *y* coordinates should be separated by a space and followed by a carriage return. For example, data values of $\text{sqr}(x)$ could be entered in a text file as

```
1 1
2 4
3 9
4 16
5 25
```

Example

Problem. Interpolate the cosine function between $x = 1x$ and $x = 20x$.

Run DIVDIF.PAS:

(K)eyboard or (F)ile entry of the data points? F

File name? SAMPLE3C.DAT

(K)eyboard or (F)ile entry of the interpolated points? K

Number of points (0-50)?15

Point 1: 1.5
Point 2: 2.5
Point 3: 3.5
Point 4: 4.5
Point 5: 5.5
Point 6: 6.5
Point 7: 7.5
Point 8: 8.5
Point 9: 9.5
Point 10: 10.5
Point 11: 11.5
Point 12: 12.5
Point 13: 13.5
Point 14: 14.5
Point 15: 15.5

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

X	Y
12.000	0.9781476
8.000	0.9902681
1.000	0.9998477
10.000	0.9848078
5.000	0.9961947
15.000	0.9659258
4.000	0.9975641
3.000	0.9986295
7.000	0.9925462
14.000	0.9702957

X	Interpolated Y value	Actual Values
1.500	9.99656668284607E-001	9.99657324975557E-001
2.500	9.99047982204853E-001	9.99048221581858E-001
3.500	9.98134846782587E-001	9.98134798421867E-001
4.500	9.96917355869352E-001	9.96917333733128E-001
5.500	9.95396200633579E-001	9.95396198367179E-001
6.500	9.93571893532269E-001	9.93571855676587E-001
7.500	9.91444906399794E-001	9.91444861373810E-001
8.500	9.89015879894104E-001	9.89015863361917E-001
9.500	9.86285623948171E-001	9.86285601537231E-001
10.500	9.83254980952454E-001	9.83254907563955E-001
11.500	9.79924765142406E-001	9.79924704620830E-001
12.500	9.76295923083642E-001	9.76296007119933E-001
13.500	9.72369781236267E-001	9.72369920397677E-001
14.500	9.68147757339141E-001	9.68147640378108E-001
15.500	9.63629212784399E-001	9.63630453208623E-001

The data is taken from a function of which the derivative could be computed exactly. Though the values in the right-hand column are not displayed on screen, they are shown here to indicate the accuracy of the routine.

Free Cubic Spline Interpolation (CUBE_FRE.INC)

Description

This example constructs a smooth curve through a given set of data points. The curve is a cubic spline interpolant with the following properties:

1. It passes through every data point.
2. It is continuous.
3. Its first derivative is continuous.
4. Its second derivative is continuous.

The second derivative is assumed to be zero at both endpoints (thus the cubic spline is “free”) of the interval determined by the data (Burden and Faires 1985, 117 ff). Cubics that join adjacent data points are of the following form:

$$S[i](x) = \text{Coef0}[i] + \text{Coef1}[i](x - x[i]) + \text{Coef2}[i](x - x[i])^2 \\ + \text{Coef3}[i](x - x[i])^3$$

where i ranges between 1 and the number of data points minus 1, the $x[i]$'s are the x -coordinates of the input data, and $x[i] \leq x < x[i + 1]$. The interpolated values of $f(x)$ are found by evaluating the i th cubic polynomial at x , where

$$x[i] \leq x \leq x[i + 1].$$

User-Defined Types

`TNvector = array[0..TNArraySize] of Real;`

Input Parameters

`NumPoints:Integer;` Number of data points
`XData:TNvector;` The x -coordinates of the data points
`YData:TNvector;` The y -coordinates of the data points
`NumInter:Integer;` Number of interpolations
`XInter:TNvector;` X -coordinates of points at which to interpolate

The preceding parameters must satisfy the following conditions:

1. X data points must be unique.
2. X data points must be in ascending order.
3. $NumPoints, NumInter \leq TNArraSize$.
4. $NumPoints > 1$.

TNArraSize fixes an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector*. *TNArraSize* is *not* a variable name and is never referenced by the procedure; hence there is no test for condition 3. If condition 3 is violated, the program will crash with an Index Out of Range error (assuming the directive `{R+}` is active).

Output Parameters

Coef0:TNvector; Coefficient of the constant term
Coef1:TNvector; Coefficient of the linear term
Coef2:TNvector; Coefficient of the squared term
Coef3:TNvector; Coefficient of the cubed term
YInter:TNvector; Interpolated values at *XInter*
Error:Byte; 0: No error
 1: X-values of the data points not unique
 2: X-values of the data points not in ascending order
 3: $NumPoints < 2$

Syntax of the Procedure Call

```
CubicSplineFree(NumPoints, XData, YData, NumInter, XInter,  
                  Coef0, Coef1, Coef2, Coef3, YInter, Error);
```

Sample Program

The sample program CUBE_FRE.PAS provides I/O functions that demonstrate the free cubic spline algorithm.

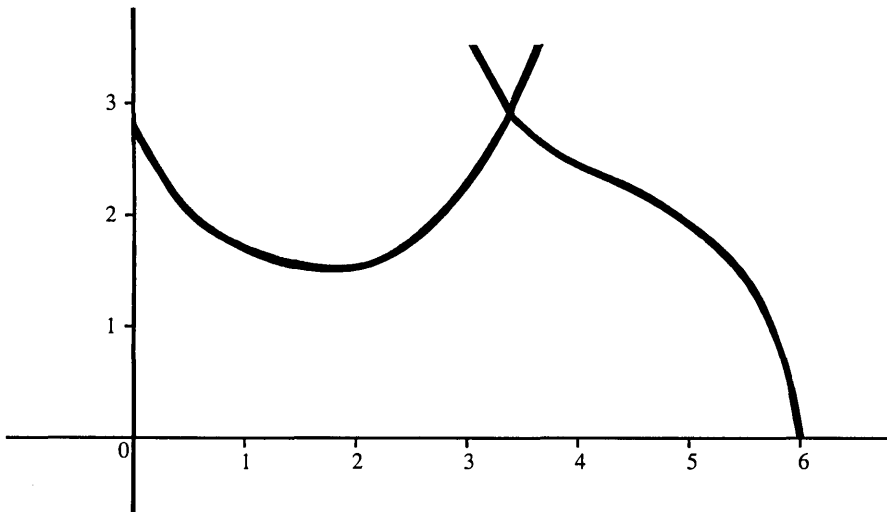
Input Files

Data may be entered from a text file. The x and y coordinates should be separated by a space and followed by a carriage return. For example, data values of $\text{sqr}(x)$ could be entered in a text file as

```
1 1
2 4
3 9
4 16
5 25
```

Example

Problem. Construct an interpolating spline for the following figure:



Because a cusp occurs at $x = 3.55$, we will construct two splines, one for each side of the cusp.

Run CUBE_FRE.PAS:

(K)eyboard or (F)ile entry of the data points? F

File name? SAMPLE3D.DAT

(K)eyboard or (F)ile entry of the interpolated points? F

File name? SAMPLE3E.DAT

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

Data:	X	Y
1:	0.0000000000	2.8000000000
2:	0.1000000000	2.7000000000
3:	0.2000000000	2.6000000000
4:	0.6000000000	2.2000000000
5:	1.0000000000	1.8000000000
6:	1.4000000000	1.6000000000
7:	1.8000000000	1.4000000000
8:	2.0000000000	1.4200000000
9:	2.2000000000	1.4000000000
10:	2.6000000000	1.5000000000
11:	3.0000000000	1.8000000000
12:	3.4000000000	2.4000000000
13:	3.4500000000	2.6000000000
14:	3.5000000000	2.8000000000
15:	3.5500000000	2.9000000000

Splines:	Coef0	Coef1	Coef2	Coef3
1:	2.800000	-0.9988332302	0.0000000000	-0.1166769808
2:	2.700000	-1.0023335396	-0.0350030942	0.5833849040
3:	2.600000	-0.9918326113	0.1400123770	-0.4010771215
4:	2.200000	-1.0723397281	-0.3412801689	1.3053237227
5:	1.800000	-0.7188084763	1.2251082984	-1.6952177695
6:	1.600000	-0.5524263669	-0.8091530249	2.3505473551
7:	1.400000	-0.0714860563	2.0115038012	-5.7703675978
8:	1.420000	0.0406713524	-1.4507167575	3.7367999767
9:	1.400000	-0.0911993534	0.7913632286	0.1540878869
10:	1.500000	0.6158534153	0.9762686929	-1.6022555777
11:	1.800000	0.6277856923	-0.9464380003	7.8174344240
12:	2.400000	3.6230038155	8.4344833084	-17.8911923822
13:	2.600000	4.3322682035	5.7508044511	-247.9233704257
14:	2.800000	3.0479233704	-31.4377011128	209.5846740851

Interpolated Points:	X	Y
1:	0.3000000000	2.5018157855
2:	0.5000000000	2.3042222482
3:	1.2000000000	1.6916808945
4:	1.6000000000	1.4759529845
5:	2.1000000000	1.4132967676
6:	2.3000000000	1.3989477848
7:	2.5000000000	1.4480232575
8:	2.7000000000	1.5697457729
9:	2.9000000000	1.7293593063
10:	3.2000000000	1.9502390938
11:	3.3000000000	2.1142270171

Second half of the figure:

(K)eyboard or (F)ile entry of the data points? F

File name? SAMPLE3F.DAT

(K)eyboard or (F)ile entry of the interpolated points? F

File name? SAMPLE3G.DAT

Direct output to one of the following:

(S)creen

(P)rinter

(F)ile

Data:	X	Y
1:	3.5500000000	2.9000000000
2:	3.6000000000	2.8000000000
3:	3.6500000000	2.6500000000
4:	3.8000000000	2.5000000000
5:	4.0000000000	2.3500000000
6:	4.3000000000	2.2000000000
7:	4.8000000000	1.9500000000
8:	5.3000000000	1.6000000000
9:	5.6000000000	1.3000000000
10:	5.8000000000	1.2000000000
11:	6.0000000000	0.0000000000

Splines:	Coef0	Coef1	Coef2	Coef3
1:	2.9000000000	-1.6719664279	0.0000000000	-131.2134288293
2:	2.8000000000	-2.6560671441	-19.6820143244	256.0671441466
3:	2.6500000000	-2.7037649955	18.7280572976	-49.1308266290
4:	2.5000000000	-0.4016786037	-3.3808146854	8.1960385189
5:	2.3500000000	-0.7704798556	1.5368084259	-2.1173630243
6:	2.2000000000	-0.4200828166	-0.3688182960	0.4179678583
7:	1.9500000000	-0.4754252188	0.2581334916	-1.4145661079
8:	1.6000000000	-1.2782163082	-1.8637156703	9.3036778805
9:	1.3000000000	0.1155473174	6.5095944222	-47.9366550462
10:	1.2000000000	-3.0330135193	-22.2523986055	37.0873310092

Interpolated Points:	X	Y
1:	3.7000000000	2.5554905401
2:	3.9000000000	2.4342200313
3:	4.1000000000	2.2862027357
4:	4.2000000000	2.2404374617
5:	4.5000000000	2.1045744477
6:	4.6000000000	2.0520666406
7:	5.0000000000	1.8539237670
8:	5.2000000000	1.7105990402
9:	5.5000000000	1.3442375346
10:	5.7000000000	1.3287140209
11:	5.9000000000	0.7112619930

Clamped Cubic Spline Interpolation (CUBE_CLA.INC)

Description

This example constructs a smooth curve through a given set of data points. The curve is a cubic spline interpolant with the following properties:

1. It passes through every data point.
2. It is continuous.
3. Its first derivative is continuous.
4. Its second derivative is continuous.

The first derivative at the endpoints of the interval determined by the input data is defined by the user (Burden and Faires 1985, 122 ff.). (This is what makes the cubic spline “clamped.”) The cubics that join adjacent data points are of the following form:

$$S[i](x) = \text{Coef0}[i] + \text{Coef1}[i](x - x[i]) + \text{Coef2}[i](x - x[i])^2 \\ + \text{Coef3}[i](x - x[i])^3$$

where i ranges between 1 and the number of data points minus 1, the $x[i]$'s are the x -coordinates of the input data, and $x[i] \leq x < x[i + 1]$. The interpolated values of $f(x)$ are found by evaluating the i th cubic polynomial at x , where $x[i] \leq x \leq x[i + 1]$.

User-Defined Types

TNvector = array[0..TNArraySize] of Real;

Input Parameters

NumPoints:Integer;	Number of data points
XData:TNvector;	The x -coordinates of the data points
YData:TNvector;	The y -coordinates of the data points
DerivLE:Real;	Derivative of the function at the left endpoint
DerivRE:Real;	Derivative of the function at the right endpoint

NumInter:Integer; Number of interpolations

XInter:TNvector; X-coordinates of points at which to interpolate

The preceding parameters must satisfy the following conditions:

1. X data points must be unique.
2. X data points must be in ascending order.
3. $NumPoints, NumInter \leq TNArrarySize$.
4. $NumPoints > 1$.

TNArrarySize fixes an upper bound on the number of elements in each vector. It is used in the *type* definition of *TNvector*. *TNArrarySize* is *not* a variable name and is never referenced by the procedure; hence there is no test for condition 3. If condition 3 is violated, the program will crash with an Index Out of Range error (assuming the directive `{R+}` is active).

Output Parameters

Coef0:TNvector; Coefficient of the constant term

Coef1:TNvector; Coefficient of the linear term

Coef2:TNvector; Coefficient of the squared term

Coef3:TNvector; Coefficient of the cubed term

YInter:TNvector; Interpolated values at *XInter*

Error:Integer; 0: No error
 1: X-values of the data points not unique
 2: X-values of the data points not in ascending order
 3: $NumPoints < 2$

Syntax of the Procedure Call

CubicSplineClamped(NumPoints, XData, YData, DerivLE, DerivRE, NumInter,
 XInter, Coef0, Coef1, Coef2, Coef3, YInter, Error);

Sample Program

The sample program CUBE_CLA.PAS provides I/O functions that demonstrate the clamped cubic spline interpolation algorithm.

Input Files

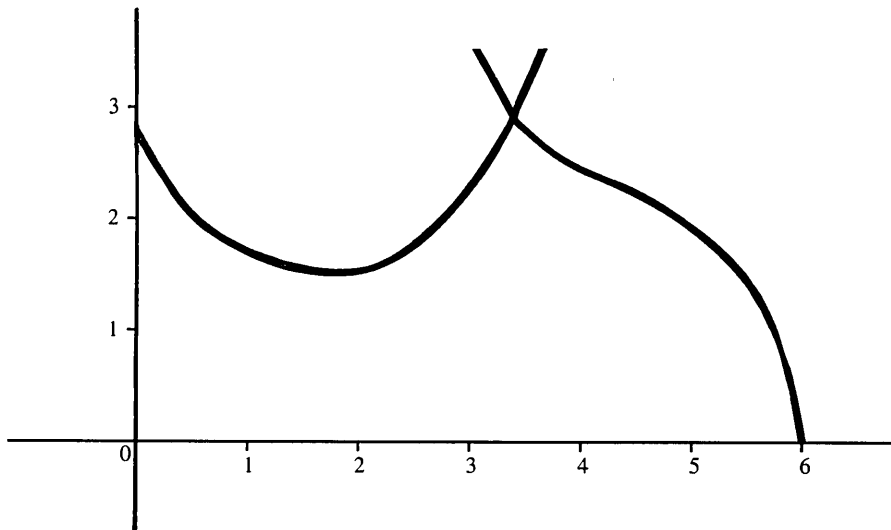
Data may be entered from a text file. The x - and y -coordinates should be separated by a space and followed by a carriage return. The last two values in the file must be the derivatives of the function at the endpoints. For example, data values of $\text{sqr}(x)$ could be entered in a text file as

```
1 1
2 4
3 9
4 16
5 25
2 10
```

Note that the last two values are the derivatives of $\text{sqr}(x)$ at the endpoints $x = 1$ and $x = 5$.

Example

Problem. Construct an interpolating spline for the following figure:



Because a cusp occurs at $x = 3.55$, we will construct two splines, one for each side of the cusp.

Run CUBE_CLA.PAS:

(K)eypress or (F)ile entry of the data points? F

File name? SAMPLE3H.DAT

(K)eypress or (F)ile entry of the interpolated points? F

File name? SAMPLE3E.DAT

Direct output to one of the following:

- (S)creen
- (P)rinter
- (F)ile

Data:	X	Y
1:	0.0000000000	2.8000000000
2:	0.1000000000	2.7000000000
3:	0.2000000000	2.6000000000
4:	0.6000000000	2.2000000000
5:	1.0000000000	1.8000000000
6:	1.4000000000	1.6000000000
7:	1.8000000000	1.4000000000
8:	2.0000000000	1.4200000000
9:	2.2000000000	1.4000000000
10:	2.6000000000	1.5000000000
11:	3.0000000000	1.8000000000
12:	3.4000000000	2.4000000000
13:	3.4500000000	2.6000000000
14:	3.5000000000	2.8000000000
15:	3.5500000000	2.9000000000

Derivative at X= 0.00000000000000E+000 : -1.33333333333333E+000

Derivative at X= 3.55000000000000E+000 : 3.00000000000000E+000

Splines:	Coef0	Coef1	Coef2	Coef3
1:	2.8000000000	-1.3333333333	5.7579845570	-24.2465122365
2:	2.7000000000	-0.9091317890	-1.5159691140	6.0728700429
3:	2.6000000000	-1.0301395105	0.3058918989	-0.5763578064
4:	2.2000000000	-1.0620777385	-0.3857374687	1.3523295373
5:	1.8000000000	-0.7215495356	1.2370579761	-1.7079603429
6:	1.6000000000	-0.5517241193	-0.8124944355	2.3545118344
7:	1.4000000000	-0.0715539872	2.0129197658	-5.7757491499
8:	1.4200000000	0.0405240212	-1.4525297241	3.7495480911
9:	1.4000000000	-0.0905420975	0.7971991306	0.1353902832
10:	1.5000000000	0.6122045428	0.9596674704	-1.5379470688
11:	1.8000000000	0.6417239262	-0.8858690121	7.5788979919
12:	2.4000000000	3.5708997526	8.2088085781	7.4639274157
13:	2.6000000000	4.4477600660	9.3283976905	-365.6719802043
14:	2.8000000000	2.6380599835	-45.5223993401	655.2239934014

Interpolated Points:	X	Y
1:	0.3000000000	2.4994686101
2:	0.5000000000	2.3029267570
3:	1.2000000000	1.6915087292
4:	1.6000000000	1.4759914934
5:	2.1000000000	1.4132766530
6:	2.3000000000	1.3990531718
7:	2.5000000000	1.4482408301
8:	2.7000000000	1.5692791819
9:	2.9000000000	1.7285068643
10:	3.2000000000	1.9535412087
11:	3.3000000000	2.1174192125

Second half of figure:

(K)eyboard or (F)ile entry of the data points? F

File name? SAMPLE3I.DAT

(K)eyboard or (F)ile entry of the interpolated points? F

File name? SAMPLE3G.DAT

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

Data:	X	Y
1:	3.5500000000	2.9000000000
2:	3.6000000000	2.8000000000
3:	3.6500000000	2.6500000000
4:	3.8000000000	2.5000000000
5:	4.0000000000	2.3500000000
6:	4.3000000000	2.2000000000
7:	4.8000000000	1.9500000000
8:	5.3000000000	1.6000000000
9:	5.6000000000	1.3000000000
10:	5.8000000000	1.2000000000
11:	6.0000000000	0.0000000000

Derivative at X= 3.550000000000000E+000 : -4.000000000000000E+000

Derivative at X= 6.000000000000000E+000 : -1.700000000000000E+001

Splines:	Coef0	Coef1	Coef2	Coef3
1:	2.9000000000	-4.0000000000	80.2233303937	-804.4666078741
2:	2.8000000000	-2.0111665197	-40.4466607874	413.3998236224
3:	2.6500000000	-2.9553339213	21.5633127559	-56.8516885392
4:	2.5000000000	-0.3238290709	-4.0199470867	9.4454622054
5:	2.3500000000	-0.7983524409	1.6473302365	-2.1760736673
6:	2.2000000000	-0.3974941891	-0.3111360640	0.2122488846
7:	1.9500000000	-0.5494435897	0.0072372629	-0.6167001671
8:	1.6000000000	-1.0047314521	-0.9178129877	3.1119483153
9:	1.3000000000	-0.7151931996	1.8829404961	-4.0348724916
10:	1.2000000000	-0.4462017001	-0.5379829989	-136.1550425028

Interpolated Points:	X	Y
1:	3.7000000000	2.5490351248
2:	3.9000000000	2.4368630843
3:	4.1000000000	2.2844619846
4:	4.2000000000	2.2388141319
5:	4.5000000000	2.1097537107
6:	4.6000000000	2.0584802174
7:	5.0000000000	1.8354671712
8:	5.2000000000	1.6919117155
9:	5.5000000000	1.3872367766
10:	5.7000000000	1.2432752125
11:	5.9000000000	1.0138449575

Numerical Differentiation

Differentiation is a process used in calculus to quantify the rate of change of a given function. The *derivative* of a real-valued function of a real variable is another real-valued function of a real variable. For example, suppose you are driving down the freeway in your car and $f(t)$ gives the distance traveled at time t . Typical values might be

x	$f(x)$
1.0	45.0
1.1	49.2
1.2	54.5
1.3	59.8
1.4	65.1
1.5	70.4

The units are in hours and miles, and the data refers to a trip that started at noon. $f(1.0) = 45.0$, so the distance traveled by one o'clock is 45.0 miles, and $f(1.5) = 70.4$, so by half past one you will be 70.4 miles from where you were at noon.

The derivative of this distance function gives the velocity function. The car's velocity at one o'clock is the value of the derivative at $x = 1.0$. From the previous data, it is impossible to compute the derivative exactly, but it is possible to approximate the derivative. The car traveled $49.2 - 45.0 = 4.2$ miles in the six minutes after one o'clock ($1.1 - 1.0 = 0.1$ hours = 6 minutes). Thus, the average velocity of the car during those six minutes is $4.2 / 0.1 = 42$ miles per hour. This gives an approximation to the velocity at one o'clock.

Each method described in this chapter approximates derivatives of a real function of one real variable.

The routines `DERIV.INC`, `DERIV2.INC`, and `INTERDRV.INC` compute derivatives of a function that is represented by tabular data. Consequently, their accuracy depends heavily upon the precision and spacing of the data points.

The routines `DERIVFN.INC` and `DERIV2FN.INC` compute derivatives of a user-defined function. Consequently, the accuracy of the values calculated with these routines is limited by the precision of the computer.

Differentiation consists of subtracting two very close numbers and dividing by a very small number; hence, it is extremely sensitive to round-off error. The accuracy of the first derivative is approximately the square root of the precision with which real numbers are represented; the accuracy of the second derivative is approximately equal to the fourth root. Thus, the precision of the first derivative will be about $1\text{E}-8$ when run with the 8087 math coprocessor, or about $1\text{E}-4$ when run without the 8087 math coprocessor. The precision of the second derivative will be about $1\text{E}-4$ with the 8087, or $1\text{E}-2$ without it.

The first derivative of a function that is represented by a table of values can be approximated in `DERIV.INC` via a two-point formula, a three-point formula, or a five-point formula. The accuracy of the formula increases with the number of points used in the formula. In order to use the five-point formula, however, the domain values of the data points (that is, the x -coordinates) must be equally spaced. This is not required for the two-point and three-point formulas. Derivatives can only be approximated at data points.

The second derivative of a function that is represented by a table of values can be approximated in `DERIV2.INC` via a three-point formula or a five-point formula. The domain values of the data points must be equally spaced (regardless of whether the three-point formula or five-point formula is used). Second derivatives can only be approximated at data points.

The routine `INTERDRV.INC` approximates a function by constructing a *free cubic spline* to a set of data points. Cubic splines avoid the undesirable oscillatory behavior of other interpolating polynomials. The derivative of the cubic spline at a given domain value, which may be different from the input data values, will then approximate the corresponding derivative of the function.

The first derivative of a user-supplied function is approximated in `DERIVFN.INC` via a three-point formula. The approximation is refined with *Richardson extrapolation*. The derivative can be approximated at any point within the domain of the function.

The second derivative of a user-supplied function is approximated in DERIV2FN.INC via a three-point formula. The approximation is refined with Richardson extrapolation. The second derivative can be approximated at any point within the domain of the function.

First Differentiation Using Two-Point, Three-Point, or Five-Point Formulas (DERIV.INC)

Description

This example contains several algorithms for approximating the derivative of a function $f(x)$, given several data points $(x, f(x))$. The user must specify whether a two-point, three-point, or five-point formula should be used. Two points are used in the two-point formula, three in the three-point formula, and five in the five-point formula. The user must supply the data points $(x, f(x))$ and the x -values of the data points at which to approximate the derivative. **Note:** Derivatives can only be approximated at x -values corresponding to input data points.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Real;
```

Input Parameters

NumPoints : Integer; Number of data points
XData : TNvector; X-coordinates of data points
YData : TNvector; Y-coordinates of data points
Point : Byte; Two-point, three-point, or five-point differentiation
NumDeriv : Integer; Number of points at which the derivative is to be approximated
XDeriv : TNvector; X-coordinates of data points at which the derivative is to be approximated

The preceding parameters must satisfy the following conditions:

1. *XData* points must be unique.
2. *XData* points must be entered in ascending order.
3. At least two points are needed for two-point differentiation, three for three-point differentiation, and five for five-point differentiation.
4. *Point* must equal two, three, or five.

5. *XData* points must be equally spaced for five-point differentiation.
6. *XDeriv* points must be a subset of the *XData* points.
7. *NumPoints*, *NumDeriv* \leq *TNArraySize*.

TNArraySize represents the number of elements in each vector. It is used in the type definition of *TNvector*. *TNArraySize* is *not* a variable name and is never referenced by the procedure; hence there is no test for condition 7. If condition 7 is violated, the program will crash with an Index Out of Range error (assuming the directive `{ $R + }` is active).

Output Parameters

YDeriv : *TNvector*; Approximation to the first derivative at the points in *XDeriv*

Error : Byte;

- 0: No errors
- 1: WARNING! Not all the derivatives were computed (see "Comments")
- 2: *X*-values not unique
- 3: *X*-values not in ascending order
- 4: Not enough data
- 5: *Point* not equal to 2, 3, or 5
- 6: *X*-values not equally spaced for the five-point formula

Syntax of the Procedure Call

`First_Derivative(NumPoints, XData, YData, Point, NumDeriv, XDeriv, YDeriv, Error);`

Comments

If an *x*-value at which the derivative is to be approximated is not among the data points, the value $-9.999999999\text{E}35$ is arbitrarily assigned to the derivative at that point and *Error* = 1 is returned. When using five-point differentiation with only five points, there is not enough information to approximate the derivative at the first, second, fourth, or fifth points. Likewise, if only six points are input, there is insufficient information for approximating the derivative at the second and fifth data points. Should an attempt be made to approximate the derivative at any of these points, the value of $9.999999999\text{E}35$ is arbitrarily assigned the derivative at that point and *Error* = 1 is returned.

Since numerical differentiation is prone to round-off errors, *TNNearlyZero* is different in this routine. The values of *TNNearlyZero* are $TNNearlyZero = 1E-13$ if using the 8087 math coprocessor and $TNNearlyZero = 1E-6$ if not using the 8087.

Sample Program

The sample program DERIV.PAS provides I/O functions that demonstrate differentiation with two-point, three-point, and five-point formulas.

Input Files

Data points may be entered from a text file. The x - and y - coordinates should be separated by a space and followed by a carriage return. For example, data values of $\text{sqr}(x)$ could be entered in a text file as

```
1 1
2 4
3 9
4 16
5 25
```

Derivative points may also be entered from a text file. Every derivative point must be followed by a carriage return. For example, to determine the derivatives of the preceding points, create the following file of derivative points:

```
1
2
3
4
5
```

Example

Problem. Approximate the first derivative of $f(x) = \text{sqr}(x) * \cos(x)$ at several points between one and two radians. The output from three runs is given. Actual values of the derivatives to eight significant figures are also given.

Run DERIV.PAS:

(K)eyboard or (F)ile entry of the data points? F

File name? SAMPLE4A.DAT

(K)eyboard or (F)ile entry of the derivative points? K

Number of X values (0-100)? 5

Point 1: 1.1

Point 2: 1.3

Point 3: 1.5

Point 4: 2.0

Point 5: 2.2

2-, 3-, or 5-point differentiation (default = 5)? 2

Direct output to one of the following:

(S)creen

(P)rinter

(F)ile

Input Data:	X	Y
	1.000000	5.40302305868140E-001
	1.100000	5.48851306924949E-001
	1.200000	5.21795166446410E-001
	1.300000	4.52073020375553E-001
	1.400000	3.33135600084472E-001
	1.500000	1.59158703752332E-001
	1.600000	-7.47507770912994E-002
	1.700000	-3.72360588514066E-001
	1.800000	-7.36134786805602E-001
	1.900000	-1.16707533637725E+000
	2.000000	-1.66458734618857E+000

Output using two-point differentiation:

```
<* ----- *>
<*          WARNING          *>
<* ----- *>
```

X	Derivative at X
1.100	8.54900105680900E-002
1.300	-6.97221460708569E-001
1.500	-1.73976896332140E+000
2.000	-4.97512009811320E+000
2.200	No derivative calculated

Output using three-point differentiation:

```
<* ----- *>
<*          WARNING          *>
<* ----- *>
```

X	Derivative at X
1.100	-9.25356971086502E-002
1.300	-9.43297831809691E-001
1.500	-2.03943188587886E+000
2.000	-5.30797739931155E+000
2.200	No derivative calculated

Output using five-point differentiation:

```
<* ----- *>
<*          WARNING          *>
<* ----- *>
```

Derivative at X		Actual Values	
X	Derivative at X	X	Derivative at X
1.100	-8.08749392678299E-002	1.100	-0.0804494
1.300	-9.32986606435738E-001	1.300	-0.9329163
1.500	-2.03221450709712E+000	1.500	-2.0321521
2.000	-5.30200229054730E+000	2.000	-5.3017771
2.200	No derivative calculated	2.200	-6.5025275

The data is taken from a function of which a derivative could be computed exactly. Though the values in the right-hand columns (under "Actual Values") are not displayed on screen, they are shown here to indicate the accuracy of the routine.

The warning signal indicates that some derivatives were not calculated.

The derivative is not approximated for $x = 2.2$ in any of the examples because $x = 2.2$ is not among the data points.

Second Differentiation Using Three-Point or Five-Point Formulas (DERIV2.INC)

Description

This example contains two algorithms that approximate the second derivative of a function $f(x)$ when several data points $(x, f(x))$ are specified. You decide whether to use a three-point or five-point formula (Gerald and Wheatley 1984, 236–237); three points are used in the three-point formula, and five in the five-point formula. You must supply the data points $(x, f(x))$ and the x -values of the data points at which the second derivative is to be approximated. The second derivative may only be approximated at x -values that were input as data points.

User-Defined Types

TNvector = array[1..TNArraySize] of Real;

Input Parameters

NumPoints : Integer; Number of data points
XData : TNvector; X-coordinates of the data points
YData : TNvector; Y-coordinates of the data points
Point : Byte; Three-point or five-point differentiation
NumDeriv : Integer; Number of points at which the derivative is to be approximated
XDeriv : TNvector; X-coordinates of points at which the derivative is to be approximated

The preceding parameters must satisfy the following conditions:

1. *XData* points must be unique.
2. *XData* points must be entered in ascending order.
3. At least three points for three-point differentiation and five points for five-point differentiation.
4. *Point* must equal 3 or 5.

5. *XData* points must be equally spaced.
6. *XDeriv* points must be a subset of the *XData* points.
7. $NumPoints, NumDeriv \leq TNArrarySize$.

TNArrarySize represents the number of elements in each vector. It is used in the **type** definition of *TNvector*. *TNArrarySize* is *not* a variable name and is never referenced by the procedure; hence there is no test for condition 7. If condition 7 is violated, the program will crash with an Index Out of Range error (assuming the directive `{ $R + }` is active).

Output Parameters

YDeriv : *TNvector*; Approximation to the second derivative at the *XDeriv* points
 Error : Byte; 0: No errors
 1: WARNING! At least one derivative was not approximated
 (see "Comments")
 2: X-values not unique
 3: X-values not in increasing order
 4: Not enough data
 5: *Point* not equal to 3 or 5
 6: X-value points not equally spaced

Syntax of the Procedure Call

`Second_Derivative(NumPoints, XData, YData, Point, NumDeriv, XDeriv, YDeriv, Error);`

Comments

If an x-value at which the second derivative is approximated is not among the data points, the value $-9.9999999E35$ is arbitrarily assigned to the derivative at that point and `Error = 1` is returned. When using five-point second differentiation with only five data points, there is insufficient information for approximating the second derivative at the second and fourth data points. Should an attempt be made to approximate the second derivative at these points, the value $9.9999999E35$ is arbitrarily assigned to the second derivative at that point and `Error = 1` is returned.

Since numerical differentiation is extremely prone to round-off error, *TNNearlyZero* is different in this routine. The values of *TNNearlyZero* are $TNNearlyZero = 1E-13$ if using the 8087 math coprocessor and $TNNearlyZero = 1E-6$ if not using the 8087.

Sample Program

The sample program DERIV2.PAS provides I/O functions that demonstrate second-order differentiation with three-point and five-point formulas.

Input Files

Data points may be entered from a text file. The x - and y -coordinates should be separated by a space and followed by a carriage return. For example, data values of $\text{sqr}(x)$ could be entered in a text file as

```
1 1
2 4
3 9
4 16
5 25
```

Derivative points may also be entered from a text file. Every derivative point must be followed by a carriage return. For example, to determine the second derivatives of the preceding points, create the following file of derivative points:

```
1
2
3
4
5
```

Example

Problem. Approximate the second derivative of $f(x) = \text{sqr}(x) * \cos(x)$ at several points between $x = 1$ and $x = 2$ radians. The output from two runs is given. Actual values of the second derivatives to eight significant figures are also given.

Run DERIV2.PAS:

(K)eyboard or (F)ile entry of the data points? F

File name? SAMPLE4A.DAT

(K)eyboard or (F)ile entry of the derivative points? K

Number of X values (0-100)?5

Point 1: 1.1

Point 2: 1.3

Point 3: 1.5

Point 4: 2.0

Point 5: 2.2

3- or 5-point second differentiation (default = 5)? 3

Direct output to one of the following:

(S)creen

(P)rinter

(F)ile

Input Data:	X	Y
	1.000000	5.40302305868140E-001
	1.100000	5.48851306924949E-001
	1.200000	5.21795166446410E-001
	1.300000	4.52073020375553E-001
	1.400000	3.33135600084472E-001
	1.500000	1.59158703752332E-001
	1.600000	-7.47507770912994E-002
	1.700000	-3.72360588514066E-001
	1.800000	-7.36134786805602E-001
	1.900000	-1.16707533637725E+000
	2.000000	-1.66458734618857E+000

Output using three-point second differentiation:

```
<* ----- *>
<*          WARNING          *>
<* ----- *>
```

X	2nd Derivative at X
1.100	-3.56051415353479E+000
1.300	-4.92152742202240E+000
1.500	-5.99325845114913E+000
2.000	-6.65714602396721E+000
2.200	No 2nd derivative calculated

Output using five-point second differentiation:

```
<* ----- *>
<*          WARNING          *>
<* ----- *>
```

		Actual Values	
X	2nd Derivative at X	X	2nd Derivative at X
1.100	-3.61167369644119E+000	1.100	-3.5629714
1.300	-4.92756964541465E+000	1.300	-4.9275779
1.500	-6.00263647117236E+000	1.500	-6.0026542
2.000	-6.59765691992321E+000	2.000	-6.4420857
2.200	No 2nd derivative calculated	2.200	-5.4434251

The data is taken from a function of which the derivative could be computed exactly. Though the values in the right-hand columns (under “Actual Values”) are not displayed on screen, they are shown here to indicate the accuracy of the routine.

The warning signal indicates that some second derivatives were not calculated.

The second derivative is not approximated at $x = 2.2$ for either run because $x = 2.2$ is not among the input x -value points.

Differentiation with a Cubic Spline Interpolant (INTERDRV.INC)

Description

This example contains an algorithm for approximating the first and second derivatives of a function given several data points $(x, f(x))$. The algorithm assumes that a free cubic spline interpolant (Burden and Faires 1985, 117–122) is an adequate approximation to the function $f(x)$, so that the slope of the interpolant at any value x_i is an adequate approximation to $f'(x_i)$. See Chapter 3 (CUBE_FRE.INC) for more information on free cubic splines. The user must supply the data points $(x, f(x))$ and the x -values at which to approximate the derivatives. Derivatives may be approximated at any x -value contained in the closed interval determined by the data points. This routine will likely give significant errors if interpolation (Gerald and Wheatley 1984, 227–231) is attempted outside the range of x -values (extrapolation).

User-Defined Types

```
TNvector = array[1..TNArraySize] of Real;
```

Input Parameters

NumPoints : Integer;	Number of data points
XData : TNvector;	X-coordinates of data points
YData : TNvector;	Y-coordinates of data points
NumDeriv : Integer;	Number of points at which the derivative is to be approximated
XDeriv : TNvector;	X-coordinates of points at which the derivative is to be approximated

The preceding parameters must satisfy the following conditions:

1. *XData* points must be unique.
2. *XData* points must be in ascending order.
3. *NumPoints* ≥ 2 .
4. *NumPoints*, *NumDeriv* \leq *TNArraySize*.

TNArraySize represents the number of elements in each vector. It is used in the type definition of *TNvector*. *TNArraySize* is *not* a variable name and is never referenced by the procedure; hence there is no test for condition 4. If condition 4 is violated, the program will crash with an Index Out of Range error (assuming the directive `{ $R+ }` is active).

Output Parameters

YInter : *TNvector*; Interpolated *y*-values at the *XDeriv* points
YDeriv : *TNvector*; Approximation to the first derivative at the *x*-values in *XDeriv*
YDeriv2 : *TNvector*; Approximation to the second derivative at the *x*-values in *XDeriv*
Error : *Byte*; 0: No errors
 1: *X*-values not unique
 2: *X*-values not in ascending order
 3: *NumPoints* < 2

Syntax of the Procedure Call

```
Interpolate_Derivative(NumPoints, XData, YData, NumDeriv,  
                      XDeriv, YInter, YDeriv, YDeriv2, Error);
```

Sample Program

The sample program `INTERDRV.PAS` provides I/O functions that demonstrate differentiation with a cubic spline interpolant.

Input Files

Data points may be entered from a text file. The x - and y -coordinates should be separated by a space and followed by a carriage return. For example, data values of $\text{sqr}(x)$ could be entered in a text file as

```
1 1
2 4
3 9
4 16
5 25
```

Derivative points may also be entered from a text file. Every derivative point must be followed by a carriage return. For example, to determine the derivatives of the preceding points, create the following file of derivative points:

```
1
2
3
4
5
```

Example

Problem. Determine the first and second derivative of $f(x) = \text{sqr}(x) * \cos(x)$ at several points between one and two radians. Actual values of the derivatives to eight significant figures are given here.

Run INTERDRV.PAS:

(K)eyboard or (F)ile entry of data points? F

File name? SAMPLE4B.DAT

(K)eyboard or (F)ile entry of derivative points? K

Number of derivative points (0-100)?5

```
Point 1: 1.1
Point 2: 1.3
Point 3: 1.55
Point 4: 1.95
Point 5: 2.20
```

Direct output to one of the following:

```
(S)creen
(P)rinter
(F)ile
```


Input Data:	X	Y
	1.000	0.5403023
	1.100	0.5488513
	1.200	0.5217952
	1.300	0.4520730
	1.400	0.3331356
	1.500	0.1591587
	1.600	-0.0747508
	1.700	-0.3723606
	1.800	-0.7361348
	1.900	-1.1670753
	2.000	-1.6645873

Using free cubic spline interpolation:

X	Value at X	1st Deriv at X	2nd Deriv at X
1.100	5.488513000000000E-001	-5.86015666816468E-002	-4.32274700
1.300	4.520730000000000E-001	-9.31377366861404E-001	-4.98862501
1.550	4.99429267146238E-002	-2.33770918101853E+000	-6.19118137
1.950	-1.41057141673716E+000	-5.01018588841893E+000	-4.20790661
2.200	-2.57545316779455E+000	-3.43222090956677E+000	16.83162644

The data is taken from a function of which the derivative could be computed exactly. The actual values are shown here:

X	Value at X	1st Deriv at X	2nd Deriv at X
1.1	0.5488513	-0.0804494	-3.5629715
1.3	0.4520730	-0.9329164	-4.9275779
1.55	0.0499596	-2.3375165	-6.2070293
1.95	-1.4076126	-4.9760746	-6.5786348
2.20	-2.8483454	-6.5025275	-5.4434252

Note the poor results obtained at values outside the range of input data ($x = 2.2$). Also note the large error in the second derivatives near the endpoints of the interval determined by the data.

Differentiation of a User-Defined Function (DERIVFN.INC)

Description

Given a user-defined function $f(x)$, this example will approximate the first derivative of the function at a set of x values. The formula

$$f'(x) = [f(x + \Delta X) - f(x - \Delta X)]/2*\Delta X$$

gives a first approximation to the derivative. Richardson extrapolation is then used to refine the approximation (Burden and Faires 1985, 137–152).

User-Defined Types

`TNvector = array[1..TNArraySize] of Real;`

User-Defined Function

`function TNTargetF(X : Real) : Real;`

Input Parameters

`NumDeriv : Integer;` Number of points at which the derivative is to be approximated

`XDeriv : TNvector;` X-coordinates of points at which the derivative is to be approximated

`Tolerance : Real;` Indicates accuracy of solution

The preceding parameters must satisfy the following conditions:

1. $NumDeriv \leq TNArraySize$
2. $Tolerance > TNNearlyZero$

TNArraySize represents the number of elements in each vector. It is used in the type definition of *TNvector*. *TNArraySize* is *not* a variable name and is never referenced by the procedure; hence there is no test for condition 1. If condition 1 is

violated, the program will crash with an Index Out of Range error (assuming the directive `{ $R+ }` is active).

Output Parameters

`YDeriv` : `TNvector`; Approximation to the first derivative at the x -values in `XDeriv`
`Error` : `Byte`; 0: No errors
 1: *Tolerance* < *TNNearlyZero*

Syntax of the Procedure Call

`FirstDerivative(NumDeriv, XDeriv, YDeriv, Tolerance, Error);`

The procedure *FirstDerivative* approximates the first derivative of function *TNTargetF*.

Comments

Since numerical differentiation is extremely prone to round-off errors, *TNNearlyZero* is different in this routine. Its values are *TNNearlyZero* = $1\text{E}-10$ if using the 8087 math coprocessor and *TNNearlyZero* = $1\text{E}-5$ if not using the 8087.

Sample Program

The sample program `DERIVFN.PAS` provides I/O functions that find the first derivative of a function at a set of points.

Input Files

Derivative points may be entered from a text file. Every derivative point must be followed by a carriage return. For example, to determine the derivatives at x -values 1 through 5, create the following file of derivative points:

1
2
3
4
5

Example

Problem. Determine the first derivative of $f(x) = \text{sqr}(x) * \cos(x)$ at several points between 1 and 2.2. Actual values of the derivatives to eight significant figures are given here.

First, write the function into the DERIVFN.PAS program:

```
(* ----- here is the function to differentiate ----- *)  
  
function TNSqrCos(X : Real) : Real;  
  
begin  
    TNSqrCos := Sqr(X)*Cos(X);  
end;                                     { function TNSqrCos }  
  
(* ----- *)
```

Run DERIVFN.PAS:

(K)eyboard or (F)ile entry of derivative points? K

Number of points (0-100)? 5

Point 1: 1.1
Point 2: 1.3
Point 3: 1.55
Point 4: 1.95
Point 5: 2.2

Tolerance (> 0, default = 1.000E-02)? 1E-4

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

Tolerance = 1.00000000000000E-004

		Actual Values		
X	Derivative at X	X	Value at X	1st Deriv at X
1.100	-8.04494385380506E-002	1.1	0.5488513	-0.0804494
1.300	-9.32916380187812E-001	1.3	0.4520730	-0.9329164
1.550	-2.33751652942968E+000	1.55	0.0499596	-2.3375165
1.950	-4.97607456093019E+000	1.95	-1.4076126	-4.9760746
2.200	-6.50252751007358E+000	2.20	-2.8483454	-6.5025275

The data is taken from a function of which the derivative could be calculated exactly. Though the values in the three right-hand columns (under "Actual Values") are not displayed on screen, they are shown here to indicate the accuracy of the routine.

Second Differentiation of a User-Defined Function (DERIV2FN.INC)

Description

Given a user-defined function $f(x)$, this example will approximate the second derivative of the function at a set of x values. The three-point formula

$$f''(x) = [f(x + \Delta X) - 2f(x) + f(x - \Delta X)]/\Delta X^2$$

gives a first approximation to the second derivative. Richardson extrapolation is then used to refine the approximation (Burden and Faires 1985, 142–152).

User-Defined Types

```
TNvector = array[1..TNArraySize] of Real;
```

User-Defined Function

```
function TNTargetF(X : Real) : Real;
```

Input Parameters

NumDeriv : Integer; Number of points at which the derivative is to be approximated

XDeriv : TNvector; X -coordinates of points at which the derivative is to be approximated

Tolerance : Real; Indicates accuracy in solution

The preceding parameters must satisfy the following conditions:

1. $NumDeriv \leq TNArraySize$
2. $Tolerance \geq TNNearlyZero$

$TNArraySize$ represents the number of elements in each vector. It is used in the type definition of $TNvector$. $TNArraySize$ is *not* a variable name and is never referenced by the procedure; hence there is no test for condition 1. If condition 1 is violated, the program will crash with an Index Out of Range error (assuming the directive $\{ \$R+ \}$ is active).

Output Parameters

$YDeriv$: $TNvector$; Approximation to the second derivative at the x -values in $XDeriv$

Error : Byte; 0: No errors
 1: $Tolerance < TNNearlyZero$

Syntax of the Procedure Call

$SecondDerivative(NumDeriv, XDeriv, YDeriv, Tolerance, Error);$

SecondDerivative approximates the derivative of function $TNTargetF$.

Comments

Since numerical differentiation is extremely prone to round-off errors, $TNNearlyZero$ is different in this routine. Its values are $TNNearlyZero = 1E-4$ if using the 8087 math coprocessor and $TNNearlyZero = 1E-2$ if not using the 8087.

Sample Program

The sample program $DERIV2FN.PAS$ provides I/O functions that find the second derivative of a function at a set of points.

Input Files

Derivative points may be entered from a text file. Every derivative point must be followed by a carriage return. For example, to determine the second derivatives at x -values 1 through 5, create the following file of derivative points:

```
1
2
3
4
5
```

Example

Problem. Determine the second derivative of $f(x) = \text{sqr}(x) * \cos(x)$ at several points between 1 and 2.2. Actual values of the derivatives to eight significant figures are given here.

First, write the function into the DERIV2FN.PAS program:

```
(* ----- here is the function to differentiate ----- *)

function TNTargetF(X : Real) : Real;

  begin
    TNTargetF := Sqr(X)*Cos(X);
  end;                                { function TNTargetF }

(* ----- *)
```

Run DERIV2FN.PAS:

(K)eyboard or (F)ile entry of derivative points? K

Number of points (0-100)? 5

```
Point 1: 1.1
Point 2: 1.3
Point 3: 1.55
Point 4: 1.95
Point 5: 2.2
```

Tolerance (> 0, default = 1.000E-02)? 1E-4

Direct output to one of the following:

```
(S)creen
(P)rinter
(F)ile
```

Tolerance = 1.000000000000000E-004

X	2nd Derivative at X	Actual Values		
		X	Value at X	2nd Deriv at X
1.100	-3.56297143915941E+000	1.1	0.5488513	-3.5629715
1.300	-4.92757787674466E+000	1.3	0.4520730	-4.9275779
1.550	-6.20702925534123E+000	1.55	0.0499596	-6.2070293
1.950	-6.57863484485542E+000	1.95	-1.4076126	-6.5786348
2.200	-5.44342524529641E+000	2.20	-2.8483454	-5.4434252

The data is taken from a function of which the derivative could be calculated exactly. Though the values in the three right-hand columns (under "Actual Values") are not displayed on screen, they are shown here to indicate the accuracy of the routine.

Numerical Integration

Integration is another concept used in calculus. It is just the opposite of differentiation, for which routines are provided in Chapter 4. Differentiation tells you the changes in a function, where integration tells you how to add those changes to get the original function.

Integration is most easily understood in terms of areas under curves. Given a function $f(x)$ and real numbers a and b with $a < b$, the area under the curve $y = f(x)$ and above the x -axis between $x = a$ and $x = b$ is given by the integral of $f(x)$ from a to b .

As with derivatives, the laws of calculus are required to compute integrals exactly. The routines in this chapter provide very accurate approximations.

Several methods are described here that approximate the value of a definite integral of a real function of one real variable. Both limits of integration must be finite.

The *trapezoid* method (TRAPZOID.INC) and *Simpson's* method (SIMPSON.INC) return an approximation of the integral when a number of equal length subintervals are specified. For a given number of subintervals, Simpson's method is preferred over the trapezoid method whenever the function being integrated is sufficiently smooth.

It is sometimes possible to approximate the definite integral to within a user-specified accuracy with fewer function evaluations using *adaptive schemes*. Adaptive schemes determine the length of each subinterval by the local behavior of the

integrand. Simpson's method (ADAPSIMP.INC) and the *Gaussian quadrature* method (ADAPGAUS.INC) are used with adaptive schemes. The Gaussian quadrature method permits, in some instances, the integrand to possess a singularity at an endpoint of integration, since the function is evaluated at points that are not the endpoints of the interval of integration.

The *Romberg* method (ROMBERG.INC) uses the trapezoid method and *Richardson extrapolation* to approximate the integral. It returns an approximation within a user-specified accuracy. Except for extremely oscillatory functions or functions that possess an endpoint singularity, this method is fastest and most accurate. If the function oscillates substantially or possesses an endpoint singularity, the adaptive Gaussian quadrature routine is preferred.

Integration Using Simpson's Composite Algorithm (SIMPSON.INC)

Description

This example uses Simpson's composite algorithm (Burden and Faires 1985, 156–167) to approximate the definite integral of a function $f(x)$ over an interval $[a, b]$. The interval is divided into N subintervals of equal length. The curve in each subinterval is approximated by a second-degree Lagrange polynomial. The integral of the resulting polynomial is then calculated. The sum of the integrals of the N Lagrange polynomials approximates the integral of the function f over the interval $[a, b]$. You must supply the function, the limits of integration, and the number of subintervals.

User-Defined Function

function TNTargetF(x : Real) : Real;

The procedure *Simpson* approximates the integral of this function.

Input Parameters

LowerLimit : Real; Lower limit of integration

UpperLimit : Real; Upper limit of integration

NumIntervals : Integer; Number of subintervals over which to apply Simpson's rule

The preceding parameters must satisfy the following condition:

NumIntervals > 0

Output Parameters

Integral : Real; Approximation to the integral of the function

Error : Byte; 0: No errors

1: $\text{NumIntervals} \leq 0$

Syntax of the Procedure Call

Simpson(LowerLimit, UpperLimit, NumIntervals, Integral, Error);

Simpson approximates the integral of *TNTargetF*.

Sample Program

The sample program SIMPSON.PAS provides I/O functions that demonstrate Simpson's composite algorithm.

Example

Problem. Approximate the integral $\exp(3x) + \text{sqr}(x)/3$ from 0 to 5 using Simpson's composite algorithm.

1. Code function *TNTargetF*:

```
function TNTargetF(x : Real) : Real;
```

```
(*****  
(***          THIS IS THE FUNCTION TO INTEGRATE          *****)  
(*****)
```

```
begin
```

```
  TNTargetF := Exp(3*X) + Sqr(X)/3;
```

```
end;                                { function TNTargetF }
```

2. Run SIMPSON.PAS:

Lower limit of integration? 0

Upper limit of integration? 5

Number of intervals (> 0)? 100

Direct output to one of the following:

(S)creen

(P)rinter

(F)ile

Lower limit: 0.00000000000000E+000

Upper limit: 5.00000000000000E+000

Number of intervals: 100

Integral: 1.08968620446199E+006

To eight significant figures, the correct answer is 1,089,686.2.

Integration Using the Trapezoid Composite Rule (TRAPZOID.INC)

Description

This example uses the trapezoid composite rule (Burden and Faires 1985, 154–167) to approximate the definite integral of a function $f(x)$ over an interval $[a, b]$. The interval is divided into N subintervals of equal length. In each subinterval the function is approximated by a straight line. The sum of the integrals of the resulting trapezoids approximates the integral of the function f over the interval $[a, b]$. You must supply the function, the limits of integration, and the number of subintervals.

User-Defined Function

```
function TNTargetF(x : Real) : Real;
```

The procedure *Trapezoid* approximates the integral of this function.

Input Parameters

Lower Limit : Real; Lower limit of integration

UpperLimit : Real; Upper limit of integration

NumIntervals : Integer; Number of subintervals over which to apply the trapezoid rule

The preceding parameters must satisfy the following condition:

NumIntervals > 0

Output Parameters

Integral : Real; Approximation to the integral of the function

Error : Byte; 0: No errors
 1: $\text{NumIntervals} \leq 0$

Syntax of the Procedure Call

Trapezoid(LowerLimit, UpperLimit, NumIntervals, Integral, Error);

Trapezoid approximates the integral of *TNTargetF*.

Sample Program

The sample program TRAPZOID.PAS provides I/O functions that demonstrate the trapezoid composite rule.

Example

Problem. Approximate the integral $\exp(3x) + \text{sqr}(x)/3$ from 0 to 5 using the trapezoid composite rule.

1. Code function *TNTargetF*:

```
function TNTargetF(x : Real) : Real;

(*****
(***      THIS IS THE FUNCTION TO INTEGRATE      *****)
(*****

begin
  TNTargetF := Exp(3*X) + Sqr(X)/3;
end;                { function TNTargetF }
```

2. Run TRAPZOID.PAS:

Lower limit of integration? 0

Upper limit of integration? 5

Number of intervals (> 0)? 100

Direct output to one of the following:

(S)creen

(P)rinter

(F)ile

Lower limit: 0.00000000000000E+000

Upper limit: 5.00000000000000E+000

Number of intervals: 100

Integral: 1.09172838320798E+006

To eight significant figures, the correct answer is 1,091,728.3.

Integration Using Adaptive Quadrature and Simpson's Rule (ADAPSIMP.INC)

Description

This example contains an algorithm for approximating the definite integral of a function $f(x)$ over an interval $[a,b]$ within a specified tolerance. By increasing the number of subintervals in regions of large functional variation (adaptive quadrature), the desired degree of accuracy can be reached (Burden and Faires 1985, 153–167). The integral within each subinterval is calculated with Simpson's rule. The adaptive quadrature approximates the integral over a subinterval twice: once over the whole subinterval, and again as the sum of the integral over each half of the subinterval. The algorithm halts when the fractional difference between these two approximations is less than the tolerance. You must supply the function, the limits of integration, and the tolerance with which to approximate the integral.

User-Defined Function

```
function TNTargetF(x : Real) : Real;
```

The procedure *Adaptive_Simpson* approximates the integral of this function.

Input Parameters

LowerLimit : Real;	Lower limit of integration
UpperLimit : Real;	Upper limit of integration
Tolerance : Real;	Indicates accuracy in solution
MaxIntervals : Integer;	Maximum number of subintervals

The preceding parameters must satisfy the following conditions:

1. *Tolerance* > 0
2. *MaxIntervals* > 0

Output Parameters

Integral : Real; Approximation to the integral of the function
NumIntervals : Integer; Number of subintervals used
Error : Byte; 0: No errors
 1: $Tolerance \leq 0$
 2: $MaxIntervals \leq 0$
 3: $NumIntervals \geq MaxIntervals$

Syntax of the Procedure Call

Adaptive_Simpson(LowerLimit, UpperLimit, Tolerance, MaxIntervals,
 Integral, NumIntervals, Error);

Adaptive_Simpson approximates the integral of *TNTargetF*.

Comments

Adaptive quadrature is a recursive routine. In order to avoid recursive procedure calls (which slow down the execution), a stack is created on the heap to simulate recursion. Should you attempt to evaluate the integral to a very high degree of accuracy with a large number of subintervals, you may get run-time error \$FF, Heap/Stack collision. If this happens, remove any RAM-resident software (for example, SideKick®, SuperKey®, or a print buffer). If the problem remains, the adaptive Simpson routine cannot be used to approximate the integral to the desired accuracy.

Adaptive quadrature uses the *New/Dispose* procedures to manipulate the heap and should not be used in any program that uses *Mark/Release* to manipulate the heap.

Sample Program

The sample program ADAPSIMP.PAS provides I/O functions that demonstrate the adaptive quadrature method with Simpson's rule.

Example

Problem. Approximate the integral $\exp(3x) + \sqrt{x}/3$ from 0 to 5 using adaptive quadrature and Simpson's rule.

1. Code function *TNTargetF*:

```
function TNTargetF(x : Real) : Real;  
  
(*****  
(**      THIS IS THE FUNCTION TO INTEGRATE      **)  
(*****)  
  
begin  
  TNTargetF := Exp(3*X) + Sqr(X)/3;  
end;      { function TNTargetF }
```

2. Run ADAPSIMP.PAS:

```
Lower limit of integration? 0  
Upper limit of integration? 5  
Tolerance (> 0, default = 1.000E-08): 1E-8  
Maximum number of subintervals (> 0, default = 1000): 1000  
Direct output to one of the following:  
  (S)creen  
  (P)rinter  
  (F)ile  
  
          Lower limit: 0.00000000000000E+000  
          Upper limit: 5.00000000000000E+000  
          Tolerance: 1.00000000000000E-008  
Maximum number of subintervals: 1000  
Number of subintervals used: 511  
  
          Integral: 1.08968601332498E+006
```

To eight significant figures, the correct answer is 1,089,686.0.

Integration Using Adaptive Quadrature and Gaussian Quadrature (ADAPGAUS.INC)

Description

This example contains an algorithm for approximating the integral of a function $f(x)$ over an interval $[a,b]$ within a specified tolerance. By increasing the number of subintervals in regions of large functional variation (adaptive quadrature), the desired degree of accuracy can be reached. The integral within each subinterval is approximated by applying Gaussian quadrature (Burden and Faires 1985, 184-188) with a 16th degree Legendre polynomial. Adaptive quadrature (Burden and Faires 1985, 172-176) approximates the integral over a subinterval twice: once over the whole subinterval, and again as the sum of the integral over each half of the subinterval. The algorithm halts when the fractional difference between these two approximations is less than the tolerance. You must supply the function, the limits of integration, and the tolerance with which to approximate the integral.

User-Defined Function

```
function TNTargetF(x : Real) : Real;
```

The procedure *Adaptive_Gauss_Quadrature* approximates the integral of this function.

Input Parameters

LowerLimit : Real;	Lower limit of integration
UpperLimit : Real;	Upper limit of integration
Tolerance : Real;	Indicates accuracy in solution
MaxIntervals : Integer;	Maximum number of subintervals

The preceding parameters must satisfy the following conditions:

1. *Tolerance* > 0
2. *MaxIntervals* > 0

The following condition is satisfied by the numbers that follow it:

Integral from -1 to 1 of $f(x) dx$

equals

Sum from $i = 1$ to *NumLegendreTerms* of

Legendre[*i*].*Weight* * $f(\text{Legendre}[i].\text{Root})$

for an arbitrary function $f(x)$.

<i>Legendre</i> [1].....	Root:	0.0950125098376370440185
	Weight:	0.189450610455068496285
<i>Legendre</i> [2].....	Root:	0.281603550778258913230
	Weight:	0.182603415044923588867
<i>Legendre</i> [3].....	Root:	0.458016777657227386342
	Weight:	0.169156519395002538189
<i>Legendre</i> [4].....	Root:	0.617876244402643748447
	Weight:	0.149595988816576732081
<i>Legendre</i> [5].....	Root:	0.755404408355003033895
	Weight:	0.124628971255533872052
<i>Legendre</i> [6].....	Root:	0.865631202387831743880
	Weight:	0.095158511682492784810
<i>Legendre</i> [7].....	Root:	0.944575023073232576078
	Weight:	0.062253523938647892863
<i>Legendre</i> [8].....	Root:	0.989400934991649932596
	Weight:	0.027152459411754094852
<i>Legendre</i> [9].....	Root:	-0.0950125098376370440185
	Weight:	0.189450610455068496285
<i>Legendre</i> [10]	Root:	-0.281603550778258913230
	Weight:	0.182603415044923588867
<i>Legendre</i> [11]	Root:	-0.458016777657227386342
	Weight:	0.169156519395002538189
<i>Legendre</i> [12]	Root:	-0.617876244402643748447
	Weight:	0.149595988816576732081
<i>Legendre</i> [13]	Root:	-0.755404408355003033895
	Weight:	0.124628971255533872052
<i>Legendre</i> [14]	Root:	-0.865631202387831743880
	Weight:	0.095158511682492784810
<i>Legendre</i> [15]	Root:	-0.944575023073232576078
	Weight:	0.062253523938647892863
<i>Legendre</i> [16]	Root:	-0.989400934991649932596
	Weight:	0.027152459411754094852

Sample Program

The sample program ADAPGAUS.PAS provides I/O functions that demonstrate the adaptive quadrature method with Gaussian quadrature.

Example

Problem. Approximate the integral $\exp(3x) + \sqrt{x}/3$ from 0 to 5 using adaptive quadrature with Gaussian quadrature algorithm.

1. Code function *TNTargetF*:

```
function TNTargetF(x : Real) : Real;
(*****
**          THIS IS THE FUNCTION TO INTEGRATE          **
*****
begin
  TNTargetF := Exp(3*X) + Sqr(X)/3;
end;                { function TNTargetF }
```

2. Run ADAPGAUS.PAS:

Lower limit of integration? 0

Upper limit of integration? 5

Tolerance (> 0, default = 1.000E-08): 1E-8

Maximum number of subintervals (> 0, default = 1000): 1000

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

```
Lower limit: 0.00000000000000E+000
Upper limit: 5.00000000000000E+000
Tolerance: 1.00000000000000E-008
Maximum number of subintervals: 1000
Number of subintervals used: 1
```

```
Integral: 1.08968601304609E+006
```

To eight significant figures, the correct answer is 1,089,686.0.

Integration Using the Romberg Algorithm (ROMBERG.INC)

Description

This example contains an algorithm (Burden and Faires 1985, 177–182) for approximating the integral of a function $f(x)$ over an interval $[a, b]$ within a specified tolerance. The trapezoid rule is used to generate a preliminary approximation, and Richardson extrapolation (Burden and Faires 1985, 148–152) is subsequently used to improve the approximation. Extrapolation continues until the fractional difference between successive approximations of the integral is less than the tolerance. You must supply the function, the limits of integration, and the tolerance with which to approximate the integral.

User-Defined Function

function TNTargetF(x : Real) : Real;

The procedure *Romberg* approximates the integral of this function.

Input Parameters

LowerLimit : Real; Lower limit of integration

UpperLimit : Real; Upper limit of integration

Tolerance : Real; Indicates accuracy in solution

MaxIter : Integer; Maximum number of iterations allowed

The preceding parameters must satisfy the following conditions:

1. *Tolerance* > 0
2. *MaxIter* > 0

Output Parameters

Integral : Real; Approximation to the integral of the function

Iter : Integer; Number of iterations

Error : Byte; 0: No errors
 1: $Tolerance \leq 0$
 2: $MaxIter \leq 0$
 3: $Iter \geq MaxIter$

Syntax of the Procedure Call

Romberg(LowerLimit, UpperLimit, Tolerance, MaxIter, Integral, Iter, Error);

Romberg approximates the integral of $TNTargetF$.

Sample Program

The sample program ROMBERG.PAS provides I/O functions that demonstrate the Romberg algorithm.

Example

Problem. Approximate the integral $\exp(3x) + \text{sqr}(x)/3$ from 0 to 5 using the Romberg algorithm.

1. Code function *TNTargetF*:

```
function TNTargetF(x : Real) : Real;

(*****
****          THIS IS THE FUNCTION TO INTEGRATE          ****
*****)
begin
  TNTargetF := Exp(3*X) + Sqr(X)/3;
end;
{ function TNTargetF }
```

2. Run ROMBERG.PAS:

Lower limit of integration? 0

Upper limit of integration? 5

Tolerance (> 0, default = 1.000E-08): 1E-8

Maximum number of iterations: (> 0, default = 100) 100

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

Lower limit: 0.00000000000000E+000

Upper limit: 5.00000000000000E+000

Tolerance: 1.00000000000000E-008

Maximum number of iterations: 100

Number of iterations: 7

Integral: 1.08968601696675E+006

To eight significant figures, the correct answer is 1,089,686.0.

Matrix Routines

This chapter provides routines for dealing with systems of linear equations. An example of a system of linear equations is as follows:

$$2X + Y + Z = 7$$

$$X - Y + Z = 2$$

$$X + Y - Z = 0$$

Matrix algebra is a collection of notations that constitutes a technique for handling such systems. With matrix algebra, the preceding system would be written

$$A x = b$$

where

$$A = \begin{bmatrix} 2 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & -1 \end{bmatrix} \quad x = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \quad b = \begin{bmatrix} 7 \\ 2 \\ 0 \end{bmatrix}$$

In Pascal, x and b are represented as one-dimensional arrays, and A is represented as a two-dimensional array. In matrix notation, the solution is given by

$$x = A^{-1} b$$

where A^{-1} is the *inverse* to A .

The determinant is an indicator of whether the matrix can be inverted. For example, the equations

$$3X - 3Y = 4$$

$$-2X + 2Y = 5$$

cannot be solved. Even for different values of the right-hand side, the equations can only be solved in certain exceptional cases. (If you change 4 and 5 to 3 and -2 , then there are infinitely many solutions; but there are none if you change 4 and 5 to 3 and -3.0001 .)

Following is a description of several routines that operate on matrices and systems of linear equations.

The determinant of a square matrix is found via DET.INC.

The inverse of a nonsingular matrix is found via INVERSE.INC.

The direct techniques implemented to solve a system of N linear equations in N unknowns are *Gaussian elimination* (GAUSELIM.INC), *Gaussian elimination with partial pivoting* (PARTPIVT.INC), and *direct factorization* (DIRFACT.INC).

The *Gauss-Seidel* method (GAUSSIDL.INC), an iterative technique that converges to the solution, is seldom used for solving small systems, since the time required for sufficient accuracy exceeds that required for the preceding direct techniques.

In general, Gaussian elimination with partial pivoting is the fastest, most accurate algorithm (see Chapter 9, LEAST.INC, for an application of PARTPIVT.INC). The following special cases may warrant the use of one of the other routines:

- If you are considering systems where round-off is minimal (that is, small systems whose coefficients are all of nearly the same magnitude), Gaussian elimination without pivoting may be used. It is somewhat faster than its pivoting counterpart (PARTPIVT.INC).
- When considering sparse coefficient matrices, the Gaussian elimination routine with partial pivoting is the most efficient and accurate routine. If the matrix is small and the nonzero coefficients do not differ wildly from each other, regular Gaussian elimination (GAUSELIM.INC) can usually be used safely.
- For large, dense matrices, the iterative technique (GAUSSIDL.INC) is the most efficient; it creates less round-off error than the direct methods. However, the Gauss-Seidel algorithm has its own weaknesses (see the section, "Solving a System of Linear Equations with the Iterative Gauss-Seidel Method," for more details).
- When it is necessary to solve several systems with the same coefficient matrix but a different vector of constant terms, the direct factorization method (DIRFACT.INC) is the most efficient. This is because it does not require reduction of the coefficient matrix for each vector of constants. (See Chapter 7 for an application of DIRFACT.INC.)

Determinant of a Matrix (DET.INC)

Description

The determinant of an $N \times N$ matrix can be computed by the following algorithm (Gerald and Wheatley 1984, 110–111):

1. Use elementary row operations to make the matrix upper triangular (that is, all the elements below the main diagonal are zero).
2. Find the product of the main diagonal elements – this will be the determinant.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Real;  
TNmatrix = array[1..TNArraySize] of TNvector;
```

Input Parameters

Dimen : Integer; Dimension of the data matrix

Data : TNmatrix; The square matrix

The preceding parameters must satisfy the following conditions:

1. $Dimen > 0$
2. $Dimen \leq TNArraySize$

TNArraySize sets an upper bound on the number of elements in each vector. It is used in the type definition of *TNvector* and *TNmatrix*. *TNArraySize* is not a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error.

Output Parameters

Det : Real; Determinant of the data matrix
Error : Byte; 0: No errors
 1: *Dimen* < 1

Syntax of the Procedure Call

Determinant(Dimen, Data, Det, Error);

Sample Program

The sample program DET.PAS provides I/O functions that demonstrate how to find the determinant of a matrix.

Input File

Data may be input from a text file. All entries in the text file should be separated by a space or carriage return, and it does not matter if the text file ends with a carriage return. The format of the text file should be like this:

1. The dimension of the matrix
2. The elements of the matrix in row order; that is,
[1, 1], [1, 2] ... [1, *N*], [2, 1] ... [2, *N*] ... [*N*, *N*],
where *N* is the dimension of the matrix

For example, a text file containing the matrix

$$\begin{bmatrix} 2 & 3 \\ -4 & 0 \end{bmatrix}$$

could look like this:

```
2
2 3
-4 0
```

Example

Problem. Find the determinant of the following matrix:

$$\begin{bmatrix} 1 & 2 & 0 & -1.0 \\ -1 & 4 & 3 & -0.5 \\ 2 & 2 & 1 & -3.0 \\ 0 & 0 & 3 & -4.0 \end{bmatrix}$$

Run DET.PAS:

(K)eyboard or (F)ile input of data? F

File name? SAMPLE6A.DAT

Direct output to one of the following:

(S)creen

(P)rinter

(F)ile

The matrix:

1.00000000	2.00000000	0.00000000	-1.00000000
-1.00000000	4.00000000	3.00000000	-0.50000000
2.00000000	2.00000000	1.00000000	-3.00000000
0.00000000	0.00000000	3.00000000	-4.00000000

Determinant = -2.10000000000000E+001

Inverse of a Matrix (INVERSE.INC)

Description

The inverse of an $N \times N$ matrix A is an $N \times N$ matrix A^{-1} , such that $A^{-1}A$ equals the identity matrix (Burden and Faires 1985, 306–316). Gauss-Jordan elimination (Gerald and Wheatley 1984, 96–98) is used to transform the original matrix into the identity matrix. The same elementary row operations that reduce A to the identity matrix transform the identity matrix into the inverse of the original matrix A . If one or more of the main diagonal elements of the transformed original matrix (that is, after Gauss-Jordan elimination) is zero, then the original matrix A is singular and its inverse does not exist.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Real;  
TNmatrix = array[1..TNArraySize] of TNvector;
```

Input Parameters

Dimen : Integer; Dimension of the data matrix

Data : TNmatrix; The elements of the square matrix

The preceding parameters must satisfy the following conditions:

1. $Dimen > 0$
2. $Dimen \leq TNArraySize$

TNArraySize fixes an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector* and *TNmatrix*. *TNArraySize* is not a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error.

Output Parameters

INV : TNmatrix; The inverse of the data matrix

Error : Byte; 0: No errors
 1: *Dimen* < 1
 2: No inverse exists

Syntax of the Procedure Call

Inverse(Dimen, Data, INV, Error);

Sample Program

The sample program INVERSE.PAS provides I/O functions that demonstrate how to find the inverse of a matrix.

Input Files

Data may be input from a text file. All entries in the text file should be separated by a space or carriage return, and it does not matter if the text file ends with a carriage return. The format of the text file should be as follows:

1. The dimension of the matrix
2. The elements of the matrix in row order; that is,
[1, 1], [1, 2] ... [1, *N*], [2, 1] ... [2, *N*] ... [*N*, *N*],
where *N* is the dimension of the matrix

For example, a text file containing the matrix

$$\begin{bmatrix} 2 & 3 \\ -4 & 0 \end{bmatrix}$$

could look like this:

```
2
2 3
-4 0
```

Example

Problem. Invert the following matrix:

$$\begin{bmatrix} 1 & 2 & 0 & -1.0 \\ -1 & 4 & 3 & -0.5 \\ 2 & 2 & 1 & -3.0 \\ 0 & 0 & 3 & -4.0 \end{bmatrix}$$

Run INVERSE.PAS:

(K)eyboard or (F)ile input of data? F

File name? SAMPLE6A.DAT

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

The matrix:

1.000000000	2.000000000	0.000000000	-1.000000000
-1.000000000	4.000000000	3.000000000	-0.500000000
2.000000000	2.000000000	1.000000000	-3.000000000
0.000000000	0.000000000	3.000000000	-4.000000000

Inverse:

-1.952380952	0.190476190	1.571428571	-0.714285714
0.761904762	0.047619048	-0.357142857	0.071428571
-1.904761905	0.380952381	1.142857143	-0.428571429
-1.428571429	0.285714286	0.857142857	-0.571428571

To continue this example, reinvert the matrix just obtained:

(K)eyboard or (F)ile input of data? F

File name? SAMPLE6B.DAT

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

The matrix:

-1.952380952	0.190476190	1.571428571	-0.714285714
0.761904762	0.047619048	-0.357142857	0.071428571
-1.904761905	0.380952381	1.142857143	-0.428571429
-1.428571429	0.285714286	0.857142857	-0.571428571

Inverse:

1.000000000	2.000000000	0.000000000	-1.000000000
-1.000000000	4.000000000	3.000000000	-0.500000000
2.000000000	2.000000000	1.000000000	-3.000000000
-0.000000000	-0.000000000	3.000000000	-4.000000000

The coefficients of the original matrix are returned to fourteen significant figures (only ten are displayed). The coefficients will be less precise if this example is run on a machine without an 8087 math coprocessor.

Solving a System of Linear Equations with Gaussian Elimination (GAUSELIM.INC)

Description

The solution to a system of N linear equations, $AX = B$, in N unknowns may be found by simultaneously performing Gaussian elimination (Burden and Faires 1985, 291–304) on the matrix containing the coefficients of the equations (the coefficient matrix A) and the vector containing the constant terms of the equations (the constant vector B). First, elementary row operations are used to make A upper triangular (that is, all the elements below the main diagonal are zero). *Backward substitution* (whereby $X[N]$ is calculated and used to calculate $X[N-1]$, which is then used to calculate $X[N-2]$, and so on) is then used to compute the solution vector X . If one or more of the elements on the main diagonal of the upper triangular matrix is zero, then no unique solution to the system exists.

User-Defined Types

`TNvector = array[1..TNArraySize] of Real;`

`TNmatrix = array[1..TNArraySize] of TNvector;`

Input Parameters

`Dimen : Integer;` Dimension of the coefficients matrix

`Coefficients : TNmatrix;` The square matrix containing the coefficients of the equations

`Constants : TNvector;` The constant terms of each equation

The preceding parameters must satisfy the following conditions:

1. $Dimen > 0$
2. $Dimen \leq TNArraySize$

TNArraySize sets an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector* and *TNmatrix*. *TNArraySize* is not a variable name and is never referenced by the procedure; hence there is no test for

condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error.

Output Parameters

Solution : TNvector; Solution to the set of equations.

Error : Byte; 0: No errors.
 1: *Dimen* < 1.
 2: Coefficients matrix is singular; no unique solution exists.

Syntax of the Procedure Call

Gaussian_Elimination(Dimen, Coefficients, Constants, Solution, Error);

Sample Program

The sample program GAUSELIM.PAS provides I/O functions that demonstrate how to solve a system of linear equations with Gaussian elimination.

Input File

Data may be input from a text file. All entries in the text file should be separated by a space or carriage return, and it does not matter if the text file ends with a carriage return. The format of the text file should be as follows:

1. The dimension of the coefficient matrix
2. The elements of the matrix in row order; that is,
[1, 1], [1, 2], ..., [1, *N*], [2, 1], ..., [2, *N*], ..., [*N*, *N*],
where *N* is the dimension of the matrix
3. The elements of the constant vector, in the order [1],..., [*N*]

For example, to solve the system

$$2x + 3y = 10$$

$$-4x = 10$$

a text file could be created to look like this:

```
2
2 3
-4 0
10
10
```

Example

Problem. Solve the following linear system:

$$w + 2x + 0y - z = 10.0$$

$$-w + 4x + 3y - 0.5z = 21.5$$

$$2w + 2x + y - 3z = 26.0$$

$$3y - 4z = 37.0$$

Run GAUSELIM.PAS:

(K)eyboard or (F)ile input of data? F

File name? SAMPLE6A.DAT

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

The coefficients:

```
1.000000000 2.000000000 0.000000000 -1.000000000
-1.000000000 4.000000000 3.000000000 -0.500000000
2.000000000 2.000000000 1.000000000 -3.000000000
0.000000000 0.000000000 3.000000000 -4.000000000
```

The constants:

```
1.000000000000000E+001
2.150000000000000E+001
2.600000000000000E+001
3.700000000000000E+001
```

The solution:

```
-1.000000000000000E+000
2.000000000000000E+000
3.000000000000000E+000
-7.000000000000000E+000
```

Solving a System of Linear Equations with Gaussian Elimination and Partial Pivoting (PARTPIVT.INC)

Description

The solution to a system of N linear equations, $AX = B$, in N unknowns may be found by simultaneously performing Gaussian elimination (Burden and Faires 1985, 291–304) on the matrix containing the coefficients of the equations (the coefficient matrix A) and the vector containing the constant terms of the equations (the constant vector B). However, excessive round-off errors can occur when elements on the main diagonal are small compared to the elements below them in the same column. To avoid this, partial pivoting (maximal column pivoting) is performed (Burden and Faires 1985, 324–327); that is, row interchanges are performed so that each main diagonal element is greater than or equal to the elements below it in the same column. (See Chapter 9 for an application of PARTPIVT.INC.)

User-Defined Types

```
TNvector = array[1..TNArraySize] of Real;  
TNmatrix = array[1..TNArraySize] of TNvector;
```

Input Parameters

Dimen : Integer; Dimension of the coefficients matrix
Coefficients : TNmatrix; The square matrix containing the coefficients of the equations
Constants : TNvector; The constant terms of each equation

The preceding parameters must satisfy the following conditions:

1. $Dimen > 0$
2. $Dimen \leq TNArraySize$

TNArraySize sets an upper bound on the number of elements in each vector. It is used in the type definition of *TNvector* and *TNmatrix*. *TNArraySize* is not a variable name and is never referenced by the procedure; hence there is no test for

condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error.

Output Parameters

Solution : TNvector; Solution to the set of equations.

Error : Byte; 0: No errors.
 1: *Dimen* < 1.
 2: Coefficients matrix is singular; no unique solution exists.

Syntax of the Procedure Call

Partial_Pivoting(Dimen, Coefficients, Constants, Solution, Error);

Sample Program

The sample program PARTPIVT.PAS provides I/O functions that demonstrate how to solve a system of linear equation with Gaussian elimination and partial pivoting.

Input File

Data may be input from a text file. All entries in the text file should be separated by a space or carriage return, and it does not matter if the text file ends with a carriage return. The format of the text file should be as follows:

1. The dimension of the matrix
2. The elements of the matrix in row order; that is,
[1, 1], [1, 2], ..., [1, *N*], [2, 1], ..., [2, *N*], ..., [*N*, *N*],
where *N* is the dimension of the matrix
3. The elements of the constant vector, in the order [1],..., [*N*]

For example, to solve the system

$$2x + 3y = 10$$

$$-4x = 10$$

a text file could be created to look like this:

```
2
2 3
-4 0
10
10
```

Example

Problem. Solve the following linear system:

$$w + 2x + 0y - z = 10$$

$$-w + 4x + 3y - 0.5z = 21.5$$

$$2w + 2x + y - 3z = 26$$

$$3y - 4z = 37$$

Run PARTPIVT.PAS:

(K)eyboard or (F)ile input of data? F

File name? SAMPLE6A.DAT

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

The coefficients:

1.000000000	2.000000000	0.000000000	-1.000000000
-1.000000000	4.000000000	3.000000000	-0.500000000
2.000000000	2.000000000	1.000000000	-3.000000000
0.000000000	0.000000000	3.000000000	-4.000000000

The constants:

1.000000000000000E+001
2.150000000000000E+001
2.600000000000000E+001
3.700000000000000E+001

The solution:

-1.000000000000000E+000
2.000000000000000E+000
3.000000000000000E+000
-7.000000000000000E+000

Solving a System of Linear Equations with Direct Factoring (DIRFACT.INC)

Description

The solution to a system of N linear equations, $AX = B$, in N unknowns can be computed by factoring the matrix containing the coefficients of the N equations (the coefficient matrix A) into an upper triangular matrix U (that is, all the elements below the main diagonal are zero) and a lower triangular matrix L (that is, all the elements above the main diagonal are zero) such that $A = LU$. Partial pivoting is used to reduce round-off error. A record of the pivoting permutations are recorded in a permutation matrix P , so that the equation is actually $A = PLU$. *Forward substitution* (analogous to backward substitution; see “Solving a System of Linear Equations with Gaussian Elimination”) is used to solve the equations $LZ = B$ (actually $LZ = PB$, where P is the pivoting permutation matrix) and $UX = Z$ (where X is the solution to the N linear equations, and Z is an intermediate solution). If the coefficient matrix cannot be factored into nonsingular triangular matrices, then no unique solution exists.

This module includes two procedures to perform this algorithm. Procedure *LU-Decompose* performs the LU decomposition of a matrix, and procedure *LU-Solve* performs forward and backward substitution to solve the linear equations. Both procedures are in the include file DIRFACT.INC.

The most efficient way to calculate the solutions to several systems with the same coefficient matrix but different constant vectors is to first decompose the coefficient matrix A into L and U (Burden and Faires 1985, 342–349). Then perform backward substitution on this decomposed matrix and each of the constant vectors B . Thus, the coefficient matrix is decomposed only once.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Real;  
TNmatrix = array[1..TNArraySize] of TNvector;
```

Procedure LU_Decompose Input Parameters

Dimen : Integer; Dimension of the coefficients matrix

Coefficients : *TNmatrix*; Square matrix containing the coefficients of the equations

The preceding parameters must satisfy the following conditions:

1. *Dimen* > 0
2. *Dimen* ≤ *TNArraySize*

TNArraySize fixes an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector* and *TNmatrix*. *TNArraySize* is not a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error.

Procedure LU_Decompose Output Parameters

Decomp : *TNmatrix*; The LU decomposition of the coefficients matrix.

Permute : *TNmatrix*; A permutation matrix that records the effects of pivoting.

Error : Byte; 0: No errors.
 1: *Dimen* < 1.
 2: The coefficients matrix is singular.

Syntax of the Procedure Call

LU_Decompose(*Dimen*, *Coefficients*, *Decomp*, *Permute*, *Error*);

Procedure LU_Solve Input Parameters

Dimen : Integer; Dimension of the coefficients matrix

Decomp : *TNmatrix*; The LU decomposition of the coefficients matrix

Constants : *TNmatrix*; The constant terms of each equation

Permute : *TNmatrix*; A permutation matrix that records the effects of pivoting

The preceding parameters must satisfy the following conditions:

1. $Dimen > 0$
2. $Dimen \leq TNArraySize$

TNArraySize fixes an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector* and *TNmatrix*. *TNArraySize* is not a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error.

Procedure LU_Solve Output Parameters

Solution : *TNvector*; Solution to each system of equations

Error : Byte; 0: No errors
 1: $Dimen < 1$

Syntax of the Procedure Call

LU_Solve(Dimen, Decom, Constants, Permute, Solution, Error);

Sample Program

The sample program DIRFACT.PAS provides I/O functions that demonstrate how to solve a system of linear equations with the method of direct factoring.

Input File

Data may be input from a text file. All entries in the text file should be separated by a space or carriage return, and it does not matter if the text file ends with a carriage return. The format of the text file should be as follows:

1. The dimension of the matrix
2. The elements of the matrix in row order; that is,
[1, 1], [1, 2], ..., [1, N], [2, 1], ..., [2, N], ..., [N, N],
where N is the dimension of the matrix
3. The elements of the first constant vector, in the order [1],..., $[N]$, with each element followed by a carriage return

4. The elements of any additional constant vectors, in the order [1],...,[N], with each element followed by a carriage return

For example, to solve the systems

$$\begin{array}{rcl} 2x + 3y & = & 10 \\ -4x & = & 10 \end{array} \qquad \begin{array}{rcl} 2x + 3y & = & 1 \\ -4x & = & 2 \end{array}$$

a text file could be created to look like this:

```
2
2 3
-4 0
10
10
1
2
```

Example

Problem. Given the following set of coefficients:

$$\begin{array}{rcl} 2w + x + 5y - 8z \\ 7w + 6x + 2y + 2z \\ -1w - 3x - 10y + 4z \\ 2w + 2w + 2y + z \end{array}$$

compute solutions for each of the five constant vectors:

$$\begin{bmatrix} 0 & -15 & 14 & -13 & 5 \\ 17 & 50 & 1 & 84 & 30 \\ -10 & -5 & -12 & -51 & -15 \\ 7 & 17 & 1 & 37 & 10 \end{bmatrix}$$

Run DIRFACT.PAS:

(K)eyboard or (F)ile input of data? F

File name? SAMPLE6C.DAT

Direct output to one of the following:

(S)creen.
(P)rinter
(F)ile

The coefficients:

2.000000000	1.000000000	5.000000000	-8.000000000
7.000000000	6.000000000	2.000000000	2.000000000
-1.000000000	-3.000000000	-10.000000000	4.000000000
2.000000000	2.000000000	2.000000000	1.000000000

The constants:

0.000000000000000E+000
1.700000000000000E+001
-1.000000000000000E+001
7.000000000000000E+000

The solution:

9.999999999999999E-001
1.000000000000000E+000
9.999999999999999E-001
9.999999999999999E-001

The constants:

-1.500000000000000E+001
5.000000000000000E+001
-5.000000000000000E+000
1.700000000000000E+001

The solution:

2.000000000000000E+000
4.999999999999999E+000
1.85358974546113E-015
3.000000000000000E+000

The constants:

1.400000000000000E+001
1.000000000000000E+000
-1.200000000000000E+001
1.000000000000000E+000

The solution:

1.000000000000000E+000
-1.000000000000000E+000
1.000000000000000E+000
-1.000000000000000E+000

The constants:

-1.300000000000000E+001
8.400000000000000E+001
-5.100000000000000E+001
3.700000000000000E+001

The solution:

3.999999999999999E+000
5.000000000000001E+000
6.000000000000000E+000
7.000000000000000E+000

The constants:

5.00000000000000E+000
3.00000000000000E+001
-1.50000000000000E+001
1.00000000000000E+001

The solution:

-1.01506105108586E-015
5.00000000000000E+000
0.00000000000000E+000
0.00000000000000E+000

Solving a System of Linear Equations with the Iterative Gauss-Seidel Method (GAUSSIDL.INC)

Description

The solution to a system of N linear equations, $AX = B$, in N unknowns can be approximated by the Gauss-Seidel iterative technique (Burden and Faires 1985, 424–432). The equation $AX = B$ is transformed into $X = TX + C$. Given an initial approximation X_o , the sequence $X_m = TX_{m-1} + C$ is generated with the following formula:

$$X_m[i] = \frac{- \sum_{j=1}^{i-1} A[i,j] X_m[j] - \sum_{j=i+1}^N (A[i,j] X_{m-1}[j]) + B[i]}{A[i,i]}$$

The algorithm halts when the fractional difference for each element of the vector X between two iterations is less than a specified tolerance.

If A is *diagonally dominant* (that is, each of the diagonal terms is greater than or equal to the sum of the off-diagonal terms in the same row), then the sequence will converge to the solution X . If the matrix A is not diagonally dominant, then the sequence may converge to the solution, but more likely it will not. You must supply the tolerance with which to approximate a solution.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Real;
```

```
TNmatrix = array[1..TNArraySize] of TNvector;
```


Input Parameters

<code>Dimen : Integer;</code>	Dimension of the coefficients matrix
<code>Coefficients : TNmatrix;</code>	The square matrix containing the coefficients of the equations
<code>Constants : TNvector;</code>	The constant terms of the equation
<code>Tol : Real;</code>	Indicates accuracy in solution
<code>MaxIter : Real;</code>	Maximum number of iterations

The preceding parameters must satisfy the following conditions:

1. $Dimen > 0$.
2. $Dimen \leq TNArraySize$.
3. $Tol > 0$.
4. $MaxIter \geq 0$.
5. The coefficients matrix may not contain a zero on the main diagonal.

TNArraySize sets an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector* and *TNmatrix*. *TNArraySize* is not a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error.

Output Parameters

<code>Solution : TNvector;</code>	Solution to the set of equations.
<code>Iter : Real;</code>	The number of iterations required to find the solution.
<code>Error : Byte;</code>	0: No errors. 1: $Iter > MaxIter$ and matrix is not diagonally dominant. 2: $Iter > MaxIter$ and matrix is diagonally dominant. 3: $Dimen < 1$. 4: $Tol \leq 0$. 5: $MaxIter < 0$. 6: Zero on the diagonal of the coefficients matrix. 7: Sequence is diverging.

If the coefficients matrix is diagonally dominant, then the Gauss-Seidel method will converge to a solution. If the coefficients matrix is not diagonally dominant, then the Gauss-Seidel may converge to a solution, but more likely it will not. Error 7 can only occur when the coefficients matrix is not diagonally dominant. If Error 1 is returned, it is likely that convergence is not possible; if Error 2 is returned, convergence is possible but will take more than *MaxIter* iterations.

If the diagonal of the coefficients matrix contains a zero (Error 6), then the Gauss-Seidel method may not be used to solve the system of equations.

If the system of equations is under-determined, the Gauss-Seidel method will still converge to a (nonunique) solution. The Gauss-Seidel method cannot distinguish between unique and nonunique solutions. If you suspect that your system of equations is under-determined, use one of the direct methods (for example, GAUSELIM.INC) to attempt a solution; Gaussian elimination will give an error if it is under-determined. Alternatively, you could use DET.INC to find the determinant; if the determinant is zero, then the system is under-determined.

Syntax of the Procedure Call

```
Gauss_Seidel(Dimen, Coefficients, Constants, Tol, MaxIter, Solution, Iter, Error);
```

Sample Program

The sample program GAUSSIDL.PAS provides I/O functions that demonstrate how to solve a system of linear equations with the iterative Gauss-Seidel method.

Input File

Data may be input from a text file. All entries in the text file should be separated by a space or carriage return, and it does not matter if the text file ends with a carriage return. The format of the text file should be as follows:

1. The dimension of the matrix
2. The elements of the matrix in row order; that is,
[1, 1], [1, 2], ..., [1, N], [2, 1], ..., [2, N], ..., [N, N],
where N is the dimension of the matrix
3. The elements of the first constant vector, in the order [1],..., [N]

For example, to solve the systems

$$20x + 3y = 10$$

$$-4y = 10$$

a text file could be created to look like this:

```
2
20    3
0    -4
10
10
```

Example

Problem. Solve the following linear system to within a tolerance of $1\text{E}-12$:

$$10v + w + 2x - 3y + 2z = -29$$

$$4v + 50w + x + z = 35$$

$$-2v + 5w - 30x + y + z = -25$$

$$6v + 4w + 10y + 3z = -46$$

$$-3v - 2w - x + 6y + 25z = -106$$

Run GAUSSIDL.PAS:

(K)eyboard or (F)ile input of data? F

File name? SAMPLE6D.DAT

Tolerance (> 0, default = 1.000E-08): 1E-12

Maximum number of iterations (> 0, default = 100): 100

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

The coefficients:

10.000000000	1.000000000	2.000000000	-3.000000000	2.000000000
4.000000000	50.000000000	1.000000000	0.000000000	1.000000000
-2.000000000	5.000000000	-30.000000000	1.000000000	1.000000000
6.000000000	4.000000000	0.000000000	10.000000000	3.000000000
-3.000000000	-2.000000000	-1.000000000	6.000000000	25.000000000

The constants:

-2.90000000000000E+001

3.50000000000000E+001

-2.50000000000000E+001

-4.60000000000000E+001

-1.06000000000000E+002

Tolerance: 1.00000000000000E-012

Maximum number of iterations: 100

Number of iterations: 15

The result:

-2.9999999999997E+000

9.9999999999999E-001

9.9999999999998E-001

-1.9999999999999E+000

-4.0000000000000E+000

Eigenvalues and Eigenvectors

The routines in this chapter can find the eigenvalues and eigenvectors. A scalar c is an *eigenvalue* (or characteristic value) of a square matrix A if there is a nonzero vector v satisfying

$$A v = c v$$

The vector v is called the *eigenvector* corresponding to c .

The eigenvalues and eigenvectors of a matrix provide a lot of information about the matrix. If a matrix is written in terms of a basis of eigenvectors, then it is *diagonal*, meaning that its only nonzero terms are on the main diagonal.

Each procedure in this chapter attempts to approximate at least one real eigenvalue (and associated eigenvector) of a real square matrix. The eigenvector is normalized so that the element with the largest magnitude is 1.

The *power* method (POWER.INC) approximates the eigenvalue that is largest in magnitude (dominant eigenvalue). The iterative process will converge slowly or not at all if the dominant eigenvalue is not simple or if it has nearly the same magnitude as the next most-dominant eigenvalue.

The *inverse power* method (INVPOWER.INC) approximates the eigenvalue nearest to a user-supplied real value. This process usually converges more rapidly than the power method, and may be used to refine the approximate value of the eigenvalue determined by the latter method (POWER.INC).

The *Wielandt* method (WIELANDT.INC) attempts to approximate a user-specified number of eigenvalues of a given matrix. The power method (POWER.INC) is first used to approximate the dominant eigenvalue of the matrix. Deflation is employed to form a deflated, square matrix (that is, a square matrix whose dimension is one less than the original matrix). The eigenvalues of the deflated matrix are identical to those of the original matrix except for the determined dominant eigenvalue. Eigenvectors of the remaining eigenvalues from the original matrix are also contained in the deflated matrix. The dominant eigenvalue of the new deflated matrix is then determined using the power method. Wielandt's method is susceptible to round-off error, thus it may be desirable to use its results as input to the inverse power method (INVPOWER.INC).

The *cyclic Jacobi* method (JACOBI.INC) approximates all the eigenvalues of a symmetric matrix. The iterative process uses orthogonal plane rotations to reduce the given matrix into a diagonal form. Although Jacobi's method is only applicable to symmetric matrices, it is much more efficient and accurate than Wielandt's method.

Real Dominant Eigenvalue and Eigenvector of a Real Matrix Using the Power Method (POWER.INC)

Description

The power method (Burden and Faires 1985, 452–456) approximates the dominant real eigenvalue of a matrix and its associated eigenvector. The dominant eigenvalue is the eigenvalue of the largest absolute magnitude. Given a square matrix A and a real nonzero vector v , a vector w is constructed by the matrix operation $Av = w$. The vector w is normalized by dividing by its element of the largest absolute magnitude q . If the absolute difference between each of the corresponding elements in w and v is less than a specified tolerance, then the procedure halts. Otherwise, v is set equal to w , and the operation repeats until a solution is found. The magnitude q is the dominant eigenvalue, and w will be the associated eigenvector of the matrix A .

You must supply the matrix A , an initial approximation to the eigenvector v , and the tolerance.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Real;  
TNmatrix = array[1..TNArraySize] of TNvector;
```

Input Parameters

Dimen : Integer;	Dimension of the matrix <i>Mat</i>
Mat : TNmatrix;	The matrix
GuessVector : TNvector;	Initial approximation to the eigenvector
MaxIter : Integer;	Maximum number of iterations
Tolerance : Real;	Indicates accuracy in solution

The preceding parameters must satisfy the following conditions:

1. $Dimen > 1$
2. $Dimen \leq TNArraSize$
3. $Tolerance > 0$
4. $MaxIter > 0$

TNArraSize fixes an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector* and *TNmatrix*. *TNArraSize* is not a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error (assuming the directive `{ $R + }` is active).

Output Parameters

Eigenvalue : Real;	Approximation to the dominant eigenvalue of the matrix
Eigenvector : TNvector;	Approximate eigenvector associated with the dominant eigenvalue
Iter : Integer;	Number of iterations required to find the solution
Error : Byte;	0: No errors 1: $Dimen \leq 1$ 2: $Tolerance \leq 0$ 3: $MaxIter \leq 0$ 4: $Iter \geq MaxIter$

Syntax of the Procedure Call

```
Power(Dimen, Mat, GuessVector, MaxIter, Tolerance,  
      Eigenvalue, Eigenvector, Iter, Error);
```

Comments

The power method will not converge if the initial approximation (*Guess*) to the eigenvector is orthogonal to the dominant eigenvector. If the initial approximation is *orthogonal*, then the power method will converge to a different eigenvector without warning. If you suspect this has happened, run the routine with several different initial approximations.

The power method may not converge to repeated eigenvalues with linearly dependent eigenvectors. Repeated eigenvalues with linearly independent eigenvectors do not pose a problem.

The eigenvectors are normalized such that the element of largest absolute magnitude in each vector is equal to one.

Sample Program

The sample program POWER.PAS provides I/O functions that demonstrate the power method of approximating eigenvalues.

Input File

Data may be input from a text file. Entries in the text file should be separated by spaces or carriage returns, and it does not matter if the text file ends with a carriage return. The format of the text file should be as follows:

1. Dimension of the matrix
2. Elements of the matrix, in the order
 $[1, 1], [1, 2], \dots, [1, N], \dots, [N, 1], \dots, [N, N]$,
 where N is the dimension of the matrix

For example, to find the dominant eigenvalue of the matrix

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

you could first create the following text file:

```
4
1
2
3
4
```

Example

Problem. Find the dominant eigenvalue of the matrix:

$$\begin{bmatrix} 2 & 10 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 4 \end{bmatrix}$$

using the initial guess (1, 2, 3).

Run POWER.PAS:

(K)eyboard or (F)ile entry of data? K

Dimension of the matrix (1-60)? 3

Matrix[1, 1]: 2
Matrix[1, 2]: 10
Matrix[1, 3]: 0
Matrix[2, 1]: 0
Matrix[2, 2]: 1
Matrix[2, 3]: 0
Matrix[3, 1]: 0
Matrix[3, 2]: 2
Matrix[3, 3]: 4

Now input an initial guess for the eigenvector:

Vector[1]: 1
Vector[2]: 2
Vector[3]: 3

Tolerance (> 0, default = 1.000E-06): 1E-8

Maximum number of iterations (> 0, default = 100): 100

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

The matrix:

2.00000000000000E+00	1.00000000000000E+01	0.00000000000000E+00
0.00000000000000E+00	1.00000000000000E+00	0.00000000000000E+00
0.00000000000000E+00	2.00000000000000E+00	4.00000000000000E+00

Tolerance: 1.00000000000000E-008

Maximum number of iterations: 100

Number of iterations: 12

The approximate eigenvector:

-2.30295155112597E-014
3.15544362088405E-030
1.00000000000000E+000

The associated eigenvalue: 4.00000000000000E+000

The exact solution is

Eigenvalue = 4

Eigenvector = (0, 0, 1)

Real Eigenvalue and Eigenvector of a Real Matrix Using the Inverse Power Method (INVPOWER.INC)

Description

Where the power method converges to the dominant real eigenvalue of a matrix (see POWER.INC), the inverse power method (Burden and Faires 1985, 459–462) converges to the real eigenvalue nearest to a user-supplied real value. Given a square matrix A , an initial approximation p to the eigenvalue, and an initial approximation v to the eigenvector, the linear system $(A - pI)w = v$ (where I is the identity matrix) is solved via LU decomposition (see Chapter 6, “Solving a System of Linear Equations with Direct Factoring”). The vector w is normalized by dividing through by the element q with the largest absolute magnitude. If the absolute difference between each of the corresponding elements in v and w is less than a specified tolerance, then the procedure halts. Otherwise, v is set equal to w , and the previous matrix equation is solved again. The process repeats until a solution is reached. The eigenvalue of A closest to p will be $(1/q + p)$, and w will be the associated eigenvector.

You must supply the matrix A , the initial approximations p and v , and the tolerance.

User-Defined Types

```
TNvector = array[1..TNarraySize] of Real;  
TNmatrix = array[1..TNarraySize] of TNvector;
```

Input Parameters

Dimen : Integer;	Dimension of the matrix <i>Mat</i>
Mat : TNmatrix;	The matrix
GuessVector : TNvector;	Initial approximation (<i>Guess</i>) of the eigenvector
ClosestVal : Real;	The approximate eigenvalue
MaxIter : Integer;	Maximum number of iterations
Tolerance : Real;	Indicates accuracy of solution

The preceding parameters must satisfy the following conditions:

1. $Dimen > 1$
2. $Dimen \leq TNArraSize$
3. $Tolerance > 0$
4. $MaxIter > 0$

TNArraSize sets an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector* and *TNmatrix*. *TNArraSize* is not a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error (assuming the directive $\{R+\}$ is active).

Output Parameters

Eigenvalue : Real;	Approximation to the eigenvalue closest to <i>ClosestVal</i>
Eigenvector : TNvector;	Approximation to the eigenvector associated with <i>Eigenvalue</i>
Iter : Integer;	Number of iterations required to find the solution
Error : Byte;	0: No errors 1: $Dimen \leq 1$ 2: $Tolerance \leq 0$ 3: $MaxIter \leq 0$ 4: $Iter \geq MaxIter$ 5: <i>Eigenvalue/Eigenvector</i> not calculated (see "Comments")

Syntax of the Procedure Call

```
InversePower(Dimen, Mat, GuessVector, ClosestVal, MaxIter,  
             Tolerance, Eigenvalue, Eigenvector, Iter, Error);
```

Comments

The inverse power method approximates the solution of a system of linear equations. If the matrix $(Mat - Eigenvalue * I)$ is singular, where I is the *identity matrix*, the method will not converge to a solution and Error 5 will be returned. If this occurs, run the routine again with a slightly different initial approximation, *ClosestVal*.

The power method may not converge to repeated eigenvalues with linearly dependent eigenvectors. Repeated eigenvalues with linearly independent eigenvectors do not pose a problem.

The inverse power method is sensitive to the initial approximation (*ClosestVal*). If *ClosestVal* is not close to an eigenvalue or lies midway between two eigenvalues, the algorithm will converge very slowly, if at all.

The eigenvectors are normalized such that the element of the largest absolute magnitude in each vector is equal to one.

Sample Program

The sample program INVPOWER.PAS provides I/O functions that demonstrate the inverse power method of approximating eigenvalues.

Input File

Data may be input from a text file. Entries in the text file should be separated by spaces or carriage returns, and it does not matter if the text file ends with a carriage return. The format of the text file should be as follows:

1. Dimension of the matrix
2. Elements of the matrix, in the order
[1, 1], [1, 2], ..., [1, N], ..., [N, 1], ..., [N, N],
where N is the dimension of the matrix
3. Elements of the initial guess, in the order [1], [2], ..., [N],
where N is the dimension of the matrix

For example, to find an eigenvalue of the matrix

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

with an initial guess of (11, 10), you could first create the following text file:

```
4
1
2
3
4
11
10
```

Example

Problem. Suppose you know that two of the eigenvalues of the matrix

$$\begin{bmatrix} 2 & 10 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 4 \end{bmatrix}$$

are approximately 1.999 and 0.7. Use the inverse power method with an initial guess of (1, 2, 3) to refine these approximations.

Run INVPOWER.PAS with 1.999 as the approximate eigenvalue:

(K)eyboard or (F)ile entry of data? K

Dimension of the matrix (1-30)? 3

```
Matrix[1, 1]: 2
Matrix[1, 2]: 10
Matrix[1, 3]: 0
Matrix[2, 1]: 0
Matrix[2, 2]: 1
Matrix[2, 3]: 0
Matrix[3, 1]: 0
Matrix[3, 2]: 2
Matrix[3, 3]: 4
```

Now input an initial guess for the eigenvector:

```
Vector[1]: 1
Vector[2]: 2
Vector[3]: 3
```

Approximate eigenvalue (default = 5.2857): 1.999

Tolerance (> 0, default = 1.000E-06): 1E-8

Maximum number of iterations (> 0, default = 200): 200

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

The matrix:

2.000000000000000E+000	1.000000000000000E+001	0.000000000000000E+000
0.000000000000000E+000	1.000000000000000E+000	0.000000000000000E+000
0.000000000000000E+000	2.000000000000000E+000	4.000000000000000E+000

Approximate eigenvalue: 1.999000000000000E+000

Tolerance: 1.000000000000000E-008

Maximum number of iterations: 200

Number of iterations: 4

The approximate eigenvector:

1.000000000000000E+000
9.12736381850482E-014
-5.13983108970145E-014

The associated eigenvalue: 2.000000000000091E+000

Run INVPOWER.PAS with 0.7 as the approximate eigenvalue:

(K)eyboard or (F)ile entry of data? K

Dimension of the matrix (1-30)? 3

Matrix[1, 1]: 2
Matrix[1, 2]: 10
Matrix[1, 3]: 0
Matrix[2, 1]: 0
Matrix[2, 2]: 1
Matrix[2, 3]: 0
Matrix[3, 1]: 0
Matrix[3, 2]: 2
Matrix[3, 3]: 4

Now input an initial guess for the eigenvector:

Vector[1]: 1
Vector[2]: 2
Vector[3]: 3

Approximate eigenvalue (default = 5.2857): 0.7

Tolerance (> 0, default = 1.000E-06): 1E-8

Maximum number of iterations (> 0, default = 200): 200

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

The matrix:

```
2.000000000000000E+000  1.000000000000000E+001  0.000000000000000E+000
0.000000000000000E+000  1.000000000000000E+000  0.000000000000000E+000
0.000000000000000E+000  2.000000000000000E+000  4.000000000000000E+000
```

Approximate eigenvalue: 7.000000000000000E-001

Tolerance: 1.000000000000000E-008

Maximum number of iterations: 200

Number of iterations: 12

The approximate eigenvector:

1.000000000000000E+000

-1.00000002395103E-001

6.66666682633328E-002

The associated eigenvalue: 9.99999976048973E-001

The exact solutions are

Eigenvalue = 2; *Eigenvector* = (1, 0, 0)

Eigenvalue = 1; *Eigenvector* = (1, -0.1, 2/30)

Real Eigenvalues and Eigenvectors of a Real Matrix Using the Power Method and Wielandt's Deflation

(WIELANDT.INC)

Description

Wielandt's deflation is a technique that approximates each real eigenvalue and related eigenvector of a matrix (Burden and Faires 1985, 452–456). Once the dominant real eigenvalue/vector of a matrix has been approximated with the power method (see “Real Dominant Eigenvalue and Eigenvector of a Real Matrix Using the Power Method”), the next most dominant real eigenvalue/vector is approximated by removing the dominant solution. This deflates the matrix. The deflated matrix will have the same eigenvalues as the original matrix (except for the removed ones). The eigenvectors of the deflated matrix will be related to the eigenvectors of the original matrix. (They will not be identical because the dimension of the deflated matrix is less than the dimension of the original matrix.) The power method then approximates the dominant eigenvalue of the deflated matrix. This process is repeated until the appropriate number (user-supplied) of eigenvalues/vectors have been approximated.

You must supply the matrix, the number of eigenvalues/vectors to approximate, and the tolerance with which to approximate the eigenvalues/vectors.

User-Defined Types

```
TNvector = array[1..TNArraySize] of Real;  
TNmatrix = array[1..TNArraySize] of TNvector;  
TNIntVector = array[1..TNArraySize] of Integer;
```

Input Parameters

Dimen : Integer;	Dimension of the matrix <i>Mat</i>
Mat : TNmatrix;	The matrix
Guess : TNvector;	Initial approximation (<i>Guess</i>) of an eigenvector

MaxEigens : Integer; Number of eigenvalues/vectors to find (at most, *Dimen*), (see “Comments”)
 MaxIter : Integer; Maximum number of iterations
 Tolerance : Real; Indicates accuracy in solution

The preceding parameters must satisfy the following conditions:

1. $Dimen > 1$
2. $Dimen \leq TNArraYSize$
3. $Tolerance > 0$
4. $MaxIter > 0$
5. $MaxEigens > 0$
6. $MaxEigens \leq Dimen$

TNArraYSize sets an upper bound on the number of elements in each vector. It is used in the **type** definition of *TNvector* and *TNmatrix*. *TNArraYSize* is not a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error (assuming the directive `{R+}` is active).

Output Parameters

NumEigens : Integer; The number of eigenvectors returned (will be $\leq MaxEigens$).
 Eigenvalues : *TNvector*; The first *NumEigens* eigenvalues of the matrix.
 Eigenvectors : *TNmatrix*; The eigenvectors associated with the eigenvalues.
 Iter : *TNIntVector*; Number of iterations required to find each eigenvalue/vector.
 Error : Byte;

- 0: No errors.
- 1: $Dimen \leq 1$.
- 2: $Tolerance \leq 0$.
- 3: $MaxIter \leq 0$.
- 4: $MaxEigens \leq 0$, $MaxEigens > Dimen$.
- 5: $Iter \geq MaxIter$.
- 6: Warning! Not a fatal error!
The last two eigenvalues aren't real.

Syntax of the Procedure Call

```
Wielandt(Dimen, Mat, Guess, MaxEigens, MaxIter, Tolerance,  
         NumEigens, Eigenvalues, Eigenvectors, Iter, Error);
```

Comments

It is often unnecessary to determine the complete eigensystem of a matrix. The parameter *MaxEigens* prevents the routine from approximating more eigenvalues/vectors than needed. For example, if the four most dominant eigenvalues of a 20×20 matrix are desired, set *MaxEigens* equal to 4. The algorithm will halt when it has approximated the four most dominant eigenvalues, thus saving a considerable amount of time. Note, however, that the dimension of the vector eigenvalues and the matrix eigenvectors must still be *TNArraySize* (that is, the same as the dimension of the matrix).

The power method may not converge to repeated eigenvalues with linearly dependent eigenvectors. Repeated eigenvalues with linearly independent eigenvectors do not pose a problem.

The eigenvectors are normalized such that the element of the largest absolute magnitude in each vector is equal to one.

This routine stores much information on the heap. If you try to compute all the eigenvalues of a large matrix (say, all 20 of a 20×20 matrix), you may get run-time error \$FF, Heap/Stack collision. If this happens, the dimension of *TNvector* and *TNmatrix* should be reduced as much as possible. If this is not possible, then remove any RAM-resident software (for example, SideKick, SuperKey, or a print buffer).

It is difficult to determine why the power method doesn't converge to a particular eigenvector; usually the eigenvalue is complex, or eigenvectors of repeated eigenvalues are linearly dependent. However, when Wielandt's deflation has deflated the matrix to a 2×2 , it is easy to determine if the eigenvalues of the 2×2 are real or complex. If the last two eigenvalues are real, then they (and their associated eigenvectors) are returned; if the last two eigenvalues are complex, Error 6 is returned. (Error 6 is only a warning; it is not a fatal error.) It is returned to give you some information about the undetermined eigenvectors.

This procedure uses the *New/Dispose* procedures to manipulate the heap and should not be used in any program that uses *Mark/Release* to manipulate the heap.

Sample Program

The sample program WIELANDT.PAS provides I/O functions that demonstrate Wielandt's method of approximating eigensystems.

Input File

Data may be input from a text file. Entries in the text file should be separated by spaces or carriage returns, and it does not matter if the text file ends with a carriage return. The format of the text file should be as follows:

1. Dimension of the matrix
2. Elements of the matrix, in the order
 $[1, 1], [1, 2], \dots, [1, N], \dots, [N, 1], \dots, [N, N]$,
where N is the dimension of the matrix

For example, to find the dominant eigenvalue of the matrix

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

you could first create the following text file:

4
1
2
3
4

Example

Problem. Find all real eigenvalues and eigenvectors of the matrix

$$\begin{bmatrix} 2 & 10 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 4 \end{bmatrix}$$

using an initial guess of (1, 2, 3).

Run Wielandt.PAS:

(K)eyboard or (F)ile entry of data? K

Dimension of the matrix (1-10)? 3

Matrix[1, 1]: 2
Matrix[1, 2]: 10
Matrix[1, 3]: 0
Matrix[2, 1]: 0
Matrix[2, 2]: 1
Matrix[2, 3]: 0
Matrix[3, 1]: 0
Matrix[3, 2]: 2
Matrix[3, 3]: 4

Now input an initial guess for the eigenvector:

Vector[1]: 1
Vector[2]: 2
Vector[3]: 3

Tolerance (> 0, default = 1.000E-06): 1E-6

Maximum number of eigenvalues/eigenvectors to find (<= 3, default = 3): 3

Maximum number of iterations (> 0, default = 200): 200

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

The matrix:

2.00000000000000E+000	1.00000000000000E+001	0.00000000000000E+000
0.00000000000000E+000	1.00000000000000E+000	0.00000000000000E+000
0.00000000000000E+000	2.00000000000000E+000	4.00000000000000E+000

Tolerance: 1.00000000000000E-006

Maximum number of eigenvalues/eigenvectors to find: 3

Maximum number of iterations: 200

Number of iterations: 10

The approximate eigenvector:

-8.32731765653921E-007
4.60590249431080E-015
1.00000000000000E+000

The associated eigenvalue: 4.00000000000004E+000

Number of iterations: 0

The approximate eigenvector:

1.00000000000000E+000
0.00000000000000E+000
0.00000000000000E+000

The associated eigenvalue: 2.0000000000000E+000

Number of iterations: 0

The approximate eigenvector:

1.0000000000000E+000

-9.9999888969117E-002

6.6666592646070E-002

The associated eigenvalue: 9.999999999991E-001

The exact solution is

Eigenvalue = 4; *Eigenvector* = (0, 0, 1)

Eigenvalue = 2; *Eigenvector* = (1, 0, 0)

Eigenvalue = 1; *Eigenvector* = (1, -0.1, 2/30)

The Complete Eigensystem of a Symmetric Real Matrix Using the Cyclic Jacobi Method (JACOBI.INC)

Description

The *eigensystem* of a symmetric matrix can be computed much more simply and efficiently than the eigensystem of an asymmetric matrix. The cyclic Jacobi method (Atkinson and Harley 1983, 154–160) is an iterative technique for approximating the complete eigensystem of a symmetric matrix to within a given tolerance. It consists of multiplying the matrix A by a series of *rotation matrices* R_i . The rotation matrices are chosen so that the elements of the upper triangular part of A (excluding the diagonal) are systematically annihilated; that is, R_1 is chosen so that $A[1, 2]$ becomes zero, R_2 is chosen so that $A[1, 3]$ becomes zero, and so on. Since the matrix is symmetric, this will also annihilate the lower triangular part of A . Because each rotation will probably change the value of elements annihilated in previous rotations, the method is iterative. Eventually, the matrix will be diagonalized. The eigenvalues will be the elements of the main diagonal of the diagonal matrix; the eigenvectors will be the corresponding rows of the matrix created by the product of the rotation matrices R_i .

User-Defined Types

```
TNvector = array[1..TNArraySize] of Real;  
TNmatrix = array[1..TNArraySize] of TNvector;
```

Input Parameters

```
Dimen : Integer;   Dimension of the matrix Mat  
Mat : TNmatrix;    The symmetric matrix  
MaxIter : Integer; Maximum number of iterations  
Tolerance : Real;   Accuracy in solution
```

The preceding parameters must satisfy the following conditions:

1. $Dimen > 1$.
2. $Dimen \leq TNArra\!y\!S\!i\!z\!e$.
3. $Tolerance > 0$.
4. $MaxIter > 0$.
5. Mat must be symmetric.

$TNArra\!y\!S\!i\!z\!e$ sets an upper bound on the number of elements in each vector. It is used in the **type** definition of $TNvector$ and $TNmatrix$. $TNArra\!y\!S\!i\!z\!e$ is not a variable name and is never referenced by the procedure; hence there is no test for condition 2. If condition 2 is violated, the program will crash with an Index Out of Range error (assuming the directive $\{ \$R+ \}$ is active).

Output Parameters

Eigenvalues : $TNvector$;	Approximation to the eigenvalues of the matrix
Eigenvectors : $TNmatrix$;	Approximation to the eigenvectors associated with the eigenvalues
Iter : Integer;	Number of iterations required to find eigenvalues/vectors
Error : Byte;	0: No errors 1: $Dimen \leq 1$ 2: $Tolerance \leq 0$ 3: $MaxIter \leq 0$ 4: Mat not symmetric 5: $Iter \geq MaxIter$

Syntax of the Procedure Call

Jacobi(Dimen, Mat, MaxIter, Tolerance, Eigenvalues, Eigenvectors, Iter, Error);

Comments

For symmetric matrices, the Jacobi method is preferred to Wielandt's deflation.

Unlike the power (POWER.INC) and inverse power (INVPOWER.INC) methods, the efficiency of the Jacobi method is not affected by repeated eigenvalues with linearly dependent eigenvectors.

The eigenvectors are normalized such that the element of largest absolute magnitude in each vector is equal to one.

Sample Program

The sample program JACOBI.PAS provides I/O functions that demonstrate Jacobi's method of approximating the eigensystem of symmetric matrices.

Input File

Data may be input from a text file. Entries in the text file should be separated by spaces or carriage returns, and it does not matter if the text file ends with a carriage return. The format of the text file should be as follows:

1. Dimension of the matrix
2. Elements of the matrix, in the order
[1, 1], [1, 2], ..., [1, N], ..., [N, 1], ..., [N, N],
where N is the dimension of the matrix

For example, to find the dominant eigenvalue of the matrix

$$\begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$$

you could first create the following text file:

```
4
1
2
2
1
```

Example

Problem. Find the complete eigensystem of the symmetric matrix

$$\begin{bmatrix} 1 & 2 & -3 & -1 \\ 2 & 1 & -1 & -3 \\ -3 & -1 & 1 & 2 \\ -1 & -3 & 2 & 1 \end{bmatrix}$$

Run JACOBI.PAS:

(K)eyboard or (F)ile entry of data? F

File name? SAMPLE7A.DAT

Tolerance (> 0, default = 1.000E-06): 1E-8

Maximum number of iterations (> 0, default = 200): 200

Direct output to one of the following:

- (S)creen
- (P)rinter
- (F)ile

The matrix:

```
1.000000000  2.000000000 -3.000000000 -1.000000000
2.000000000  1.000000000 -1.000000000 -3.000000000
-3.000000000 -1.000000000  1.000000000  2.000000000
-1.000000000 -3.000000000  2.000000000  1.000000000
```

Tolerance: 1.000000000000000E-008

Maximum number of iterations: 200

Number of iterations: 4

The approximate eigenvector:

```
1.000000000000000E+000
1.000000000000000E+000
-1.000000000000000E+000
-1.000000000000000E+000
```

The associated eigenvalue: 7.000000000000000E+000

The approximate eigenvector:

```
-9.9999999977159E-001
9.9999999977775E-001
1.000000000000000E+000
-9.999999999384E-001
```

The associated eigenvalue: 1.000000000000000E+000

The approximate eigenvector:

```
1.000000000000000E+000
-9.9999556935431E-001
9.9999999977774E-001
-9.9999556913205E-001
```

The associated eigenvalue: $-2.9999999999999990E+000$

The approximate eigenvector:

$9.99999556935431E-001$

$9.999999999999384E-001$

$9.99999556934815E-001$

$1.000000000000000E+000$

The associated eigenvalue: $-1.000000000000010E+000$

The exact solution is

Eigenvalue = 7; *Eigenvector* = (1, 1, -1, -1)

Eigenvalue = 1; *Eigenvector* = (-1, 1, 1, -1)

Eigenvalue = -3; *Eigenvector* = (1, -1, 1, -1)

Eigenvalue = -1; *Eigenvector* = (1, 1, 1, 1)

Initial Value and Boundary Value Methods

A *differential equation* is like an ordinary equation except that the unknown is a function, and derivatives of the function appear in the equation. For example,

$$f''(x) + f(x) = 0$$

is a differential equation. $f''(x)$ is the second derivative of $f(x)$. The solutions are the functions of the form

$$f(x) = a * \cos(x) + b * \sin(x)$$

The function is uniquely determined by suitable initial conditions, such as

$$f(0) = 3$$

$$f'(0) = 4$$

in which case the solution is

$$f(x) = 3 * \cos(x) + 4 * \sin(x)$$

The routines in this chapter solve differential equations that are *ordinary* and *linear*. A differential equation is ordinary if there is only an independent variable (that is, the unknown function is a function of only one variable), and thus the derivatives are ordinary derivatives and not partial derivatives. A differential equation is linear if the unknown function and its derivatives appear linearly in the equation.

This chapter describes routines that specifically solve: (1) initial value problems for n th-order ordinary differential equations, (2) initial value problems for systems of coupled first-order and second-order ordinary differential equations, and (3)

boundary value problems for second-order ordinary differential equations.

Note that these routines work only with ordinary differential equations, not partial differential equations. All of the routines in this chapter can solve problems involving nonlinear equations (except the linear-shooting routine LINSHOT 2.INC).

Two one-step techniques that solve initial value problems for first-order ordinary differential equations are implemented. The first technique employs the *fourth-order Runge-Kutta* method (RUNGE_1.INC), also known as the *classical Runge-Kutta* method. The second employs the *Runge-Kutta-Fehlberg* method (RKF_1.INC).

Each one-step technique approximates the value of the dependent variable at a *mesh point*, which is a value of the independent variable, by using only the information obtained from the preceding mesh point. The Runge-Kutta method employs equally spaced mesh points. On the other hand, the Runge-Kutta-Fehlberg method varies the spacing of the mesh points in order to control the local truncation error. This produces a corresponding bound on the global error.

The *Adams-Bashforth/Adams-Moulton predictor/corrector* method (ADAMS_1.INC) is a multistep method that uses information obtained at several preceding mesh points to approximate the value of the dependent variable at the current mesh point. The procedure employs the Adams-Bashforth four-step method to obtain a predictor. It is subsequently used as input for the Adams-Moulton three-step method to obtain a corrector. The corrector is the approximate value of the solution. Mesh points are equally spaced, and the starting values for the process are determined by the one step, fourth-order Runge-Kutta method.

The Runge-Kutta methods are the most reliable and should be used when you are uncertain of the behavior of the differential equation (for example, if the solution to the differential equation is not very smooth). If you want the output to be evenly spaced (in x) or do not require a high degree of accuracy, use the classical Runge-Kutta method. Otherwise, the Runge-Kutta-Fehlberg method is the best general purpose routine to use, since it provides control over the accuracy of the solution.

The Adams-Bashforth/Adams-Moulton method achieves the same accuracy (for equally spaced mesh points) as the fourth-order Runge-Kutta formula, but it is significantly faster. Consequently, the Adams-Bashforth/Adams-Moulton method is the most desirable method if you are reasonably certain that the differential equation is well-behaved.

Initial value problems for first-order ordinary differential equations are guaranteed to have a unique solution on the interval a, b if the function

$$x' = f(t, x)$$

is continuous over the interval a, b , and if the function satisfies the *Lipshitz condition*. The Lipshitz condition states that there exists a positive number L such that

$$|f(t, x_2) - f(t, x_1)| \leq L|x_2 - x_1|$$

for all $a \leq t \leq b$, $-\infty < x < \infty$.

Initial value problems for second-order ordinary differential equations can be solved via a fourth-order Runge-Kutta method (RUNGE_2.INC). This procedure reduces a given differential equation to a system of two, first-order ordinary differential equations. The solution to this system is approximated at equally spaced mesh points with the fourth-order Runge-Kutta method.

Initial value problems for second-order ordinary differential equations are guaranteed to have a unique solution on the interval a, b if the function

$$x'' = f(t, x, x')$$

is continuous over the interval a, b and if the function satisfies the Lipshitz condition. For a second-order differential equation, the Lipshitz condition states that there exists a positive number L such that

$$|f(t, x_2, x'_2) - f(t, x_1, x'_1)| \leq L(|x_2 - x_1| + |x'_2 - x'_1|)$$

for all $a \leq t \leq b$, $-\infty < x < \infty$, $-\infty < x' < \infty$.

The Runge-Kutta method can be generalized for any order ordinary differential equation. The file RUNGE_N.INC contains an algorithm that can solve an initial value problem for an n th-order differential equation with the fourth-order Runge-Kutta formulas. The Lipshitz condition can be generalized for any order ordinary differential equation. (For details, consult the reference book listed in the section, "Solution to an Initial Value Problem for a First-Order Ordinary Differential Equation Using the Runge-Kutta Method.")

Although RUNGE_N.INC can be used to solve initial value problems for first-order and second-order ordinary differential equations, we recommend that RUNGE_1.INC and RUNGE_2.INC be used instead. The notation used by these routines is somewhat simpler than the general case. There is no significant difference in computation time between the general program (RUNGE_N.INC) and the specific programs (RUNGE_1.INC and RUNGE_2.INC).

Systems of coupled equations may also be solved with Runge-Kutta techniques. A system of up to ten first-order ordinary differential equations can be solved with the file RUNGE_S1.INC. A system of up to ten second-order ordinary differential equations can be solved with the file RUNGE_S2.INC. The algorithms in both these files are based on the classical Runge-Kutta method with uniform spacing between mesh points; hence, they do not allow for accuracy control (as in the Runge-Kutta-Fehlberg method). (The Lipshitz condition for systems of equations is given in the reference in the sections about RUNGE_S1.INC and RUNGE_S2.INC.)

Boundary value problems for second-order ordinary differential equations (where the value of the dependent variable is specified at the two endpoints of interval) can be solved using *shooting techniques*. Shooting techniques converge onto the slope of the function at one boundary. This reduces the boundary value problem to a series of initial value problems. The series concludes when the initial value problem satisfies the boundary condition at the other boundary.

If the second-order differential equation is linear (that is, linear in the *dependent* variable(s), not necessarily linear in the *independent* variable), the linear-shooting method (LINSHOT2.INC) may be used. A linear combination of solutions to two initial value problems yields the solution to the boundary value problem.

If the second-order differential equation is nonlinear, the routine SHOOT2.INC must be used. The secant method generates a sequence of solutions with different values of the first derivative until the appropriate boundary condition, subject to a desired accuracy, is satisfied. Although SHOOT2.INC may be used to solve linear boundary value problems, LINSHOT2.INC is more efficient for the linear case.

Boundary value problems for second-order differential equations are guaranteed to have a unique solution on the interval a, b if the function

$$y'' = f(x, y, y')$$

and the two partial derivatives $\partial f/\partial y$, $\partial f/\partial y'$ are continuous on the interval $[a, b]$. Furthermore, $\partial f/\partial y$ must be positive and $\partial f/\partial y'$ must be bounded for all x, y, y' $a \leq x \leq b$, $-\infty < y < \infty$, $-\infty < y' < \infty$.

The convergence to the appropriate initial value of the first derivative is not assured for nonlinear boundary value problems. A good guess of the derivative boundary condition is often required and may involve considerable trial and error.

Interpolation techniques (see Chapter 3) may be used to approximate the solution of values of the independent variable that are not mesh points.

Solution to an Initial Value Problem for a First-Order Ordinary Differential Equation Using the Runge-Kutta Method (RUNGE_1.INC)

Description

This example uses the Runge-Kutta method (Burden and Faires 1985, 220–227) to approximate the solution to a first-order ordinary differential equation with a specified initial condition.

Given a function of the form

$$dx/dt = TNTargetF(t, x)$$

which satisfies the conditions given at the beginning of this chapter, and an initial condition

$$x[LowerLimit] = XInitial$$

and spacing

$$h = (UpperLimit - LowerLimit)/NumIntervals$$

the fourth-order Runge-Kutta method approximates x in the interval $[LowerLimit, UpperLimit]$.

The fourth-order Runge-Kutta formulas consist of the following:

$$\begin{aligned} F1 &= h * TNTargetF(t, x[t]) \\ F2 &= h * TNTargetF(t + h/2, x[t] + F1/2) \\ F3 &= h * TNTargetF(t + h/2, x[t] + F2/2) \\ F4 &= h * TNTargetF(t + h, x[t] + F3) \\ x[t + 1] &= x[t] + (F1 + 2 * F2 + 2 * F3 + F4)/6 \end{aligned}$$

where t ranges from $LowerLimit$ to $UpperLimit$ in steps of h . These formulas give a truncation error of order h^4 .

You must supply $LowerLimit$, $UpperLimit$, $XInitial$, $NumIntervals$, and $TNTargetF$.

User-Defined Types

```
TNvector = array[1..TNarraySize] of Real;
```

User-Defined Function

TNTargetF(*t*, *X* : Real) : Real;

$$dx/dt = \text{TNTargetF}(t, x)$$

The function *TNTargetF*(*t*, *x*) is a user-defined function that calculates the derivative *dx/dt*.

Input Parameters

LowerLimit : Real; Lower limit of interval

UpperLimit : Real; Upper limit of interval

XInitial : Real; Value of *X* at *LowerLimit*

NumReturn : Integer; Number of (*t*, *x*) pairs returned from the procedure

NumIntervals : Integer; Number of subintervals used in calculations

The preceding parameters must satisfy the following conditions:

1. *NumReturn* > 0
2. *NumIntervals* ≥ *NumReturn*
3. *LowerLimit* ≠ *UpperLimit*

Output Parameters

TValues : TNvector; Values of *t* between the limits

XValues : TNvector; Values of *X* approximated at the values in *TValues*

Error : Byte; 0: No errors

1: *NumReturn* < 1

2: *NumIntervals* < *NumReturn*

3: *LowerLimit* = *UpperLimit*

Syntax of the Procedure Call

```
InitialCond1stOrder(LowerLimit, UpperLimit, XInitial, NumReturn,  
                    NumIntervals, TValues, XValues, Error);
```

The procedure *InitialCondition1stOrder* integrates the first-order differential equation.

Comments

This procedure will compute *NumIntervals* values in its calculations; however, you will rarely need to use all the values. The vectors *TValues* and *XValues* will contain only *NumReturn* values at roughly equally spaced *t*-values between the lower and upper limits. (They will be equally spaced only when *NumIntervals* is a multiple of *NumReturn*.) Thus, you can ensure a highly accurate solution (by making *NumIntervals* large) without generating an excessive amount of output (by making *NumReturn* small).

The Runge-Kutta method uses the *New/Dispose* procedures to manipulate the heap and should not be used in any program that uses *Mark/Release* to manipulate the heap.

Warning: A *stiff differential equation* occurs when there are at least two very different scales of the independent variable on which the dependent variable(s) is changing; for example, $y = x + e^{-100x}$. The Runge-Kutta method may generate a numerical solution that bears no resemblance to the exact solution of the differential equation. This unstable numerical solution usually grows exponentially and may be oscillatory. However, if the exact solution of the differential equation grows as the independent variable increases, the instability may be difficult to detect. If a suspected instability has been encountered, reduce the interval size (*NumIntervals*).

Sample Program

The sample program RUNGE_1.PAS provides I/O functions that demonstrate the Runge-Kutta method of solving initial value problems. Note that the file RUNGE_1.INC is included after the function *TNTargetF* is defined.

Example

Problem. Solve the following initial value problem with the Runge-Kutta method:

$$\begin{aligned}x' &= x/t + t - 1 & 1 \leq t \leq 2 \\x(1) &= 1\end{aligned}$$

1. Code the equation into the program RUNGE_1.PAS:

```
function TNTargetF(t, X : Real) : Real;

(*****
****          THIS IS THE FIRST-ORDER DIFFERENTIAL EQUATION          ****
*****)

begin
  TNTargetF := x/t + t - 1
end;                                { function TNTargetF }
```

2. Run RUNGE_1.PAS:

```
Lower limit of interval? 1
Upper limit of interval? 2
X value at t = 1.00000000E+00: 1
Number of values to return (1-500)? 10
Number of intervals (>= 10, default = 10)? 100
Direct output to one of the following:
(S)creen
(P)rinter
(F)ile

      Lower limit: 1.00000000000000E+000
      Upper limit: 2.00000000000000E+000
Value of X at 1.0000: 1.00000000000000E+000
Number of intervals: 100
```

t	X
1.00000000	1.00000000000000E+000
1.10000000	1.10515880220649E+000
1.20000000	1.22121413182916E+000
1.30000000	1.34892645616477E+000
1.40000000	1.48893886869362E+000
1.50000000	1.64180233779216E+000
1.60000000	1.80799419315265E+000
1.70000000	1.98793197313186E+000
1.80000000	2.18198400310574E+000
1.90000000	2.39047761619428E+000
2.00000000	2.61370563879444E+000

The exact solution is

$$\begin{aligned}X &= t^2 - t * \ln(t) \\X(2) &= 2.6137056\end{aligned}$$

Solution to an Initial Value Problem for a First-Order Ordinary Differential Equation Using the Runge-Kutta-Fehlberg Method (RKF_1.INC)

Description

This example uses the Runge-Kutta-Fehlberg method (Burden and Faires 1985, 230–235) to approximate a solution within a specified tolerance to a first-order ordinary differential equation with a specified initial condition.

Where the Runge-Kutta method (see RUNGE_1.INC) uses a constant spacing h , the Runge-Kutta-Fehlberg method varies the spacing so that the solution can be approximated with accuracy.

Given a function of the form

$$dx/dt = \text{TNTargetF}(t, x)$$

which satisfies the conditions given at the beginning of this chapter, and an initial condition

$$x[\text{LowerLimit}] = X\text{Initial}$$

both the fourth-order and fifth-order Runge-Kutta formulas are used to approximate x in the interval $[\text{LowerLimit}, \text{UpperLimit}]$. The number of subintervals is continually increased until the fractional difference between the results of the fourth-order and fifth-order formulas (which give a truncation error of h^4 and h^5 , respectively) in each subinterval is less than the specified tolerance.

You must supply *LowerLimit*, *UpperLimit*, *Tolerance*, and *TNTargetF*.

User-Defined Types

`TNvector = array[1..TNArraySize] of Real;`

User-Defined Function

`TNTargetF(t, X : Real) : Real;`

$$dx/dt = \text{TNTargetF}(t, x)$$

Input Parameters

`LowerLimit` : Real; Lower limit of interval
`UpperLimit` : Real; Upper limit of interval
`XInitial` : Real; Value of X at *LowerLimit*
`Tolerance` : Real; Maximum tolerable fractional difference between iterate values
`NumReturn` : Integer; Number of (t, x) values to be returned

The preceding parameters must satisfy the following conditions:

1. $Tolerance > 0$
2. $NumReturn > 0$
3. $LowerLimit \neq UpperLimit$

Output Parameters

`TValues` : `TNvector`; Values of t at which X was approximated
`XValues` : `TNvector`; Values of X at the values in *TValues*
`Error` : Byte;
 0: No errors
 1: $Tolerance \leq 0$
 2: $NumReturn \leq 0$
 3: $LowerLimit = UpperLimit$
 4: $Tolerance$ not reached

Syntax of the Procedure Call

`RungeKuttaFehlberg`(`LowerLimit`, `UpperLimit`, `XInitial`, `Tolerance`,
 `NumReturn`, `TValues`, `XValues`, `Error`);

The procedure *RungeKuttaFehlberg* integrates the first-order differential equation *TNTargetF*.

Comments

This procedure will compute more values in its calculations than it will return in the vectors *TValues* and *XValues*. The vectors *TValues* and *XValues* will contain only *NumReturn* values at subintervals between the lower and upper limits. More values will be returned in regions of large functional variation than in regions of small functional variation. Thus, you can ensure a highly accurate solution (by making the *Tolerance* small) without generating an excessive amount of output (by making *NumReturn* small).

The Runge-Kutta-Fehlberg method improves the accuracy in the solution by reducing the spacing between successive values of t . However, if the *Tolerance* is too small, the spacing required to reach *Tolerance* may be beyond the machine's limit of precision. Consequently, the routine will not converge to a solution that meets the required *Tolerance* and Error 5 will be returned.

The Runge-Kutta-Fehlberg method uses the *New/Dispose* procedures to manipulate the heap and should not be used in any program that uses *Mark/Release* to manipulate the heap.

Warning: A stiff differential equation occurs when there are at least two very different scales of the independent variable on which the dependent variable(s) is changing; for example, $y = x + e^{-100x}$. The Runge-Kutta-Fehlberg method may generate a numerical solution that bears no resemblance to the exact solution of the differential equation. This unstable numerical solution usually grows exponentially and may be oscillatory. However, if the exact solution of the differential equation grows as the independent variable increases, the instability may be difficult to detect. If a suspected instability has been encountered, reduce the interval size (*NumIntervals*).

Sample Program

The sample program RKF_1.PAS provides I/O functions that demonstrate the Runge-Kutta-Fehlberg method of solving initial value problems. Note that the file RKF_1.INC is included after the function *TNTargetF* is defined.

Example

Problem. Use the Runge-Kutta-Fehlberg method to solve the following initial value problem with a tolerance of 1E-6:

$$\begin{aligned}x' &= x/t + t - 1 & 1 \leq t \leq 2 \\ x(1) &= 1\end{aligned}$$

1. Code the differential equation into the program RKF_1.PAS:

```
function TNSolveF(t, X : Real) : Real;

(*****
****      THIS IS THE FIRST-ORDER DIFFERENTIAL EQUATION      ****
*****)

begin
  TNSolveF := x/t + t - 1;
end;                                { function TNSolveF }
```

2. Run RKF_1.PAS:

Lower limit of interval? 1

Upper limit of interval? 2

X value at t = 1.0000000E+00: 1

Number of values to return (1-500)? 10

Tolerance (> 0, default = 1.000E-06)? 1E-6

Direct output to one of the following:

(S)screen
(P)printer
(F)file

Lower limit: 1.0000000000000E+000
Upper limit: 2.0000000000000E+000
Value of X at 1.0000: 1.0000000000000E+000
Tolerance: 1.0000000000000E-006

t	X
1.00000000	1.0000000000000E+000
1.10000000	1.10515881708653E+000
1.20000000	1.22121416069278E+000
1.30000000	1.34892649817459E+000
1.40000000	1.48893892310351E+000
1.50000000	1.64180240395245E+000
1.60000000	1.80799427050390E+000
1.70000000	1.98793206119471E+000
1.80000000	2.18198410146987E+000
1.90000000	2.39047772450816E+000
2.00000000	2.61370575675625E+000

Now solve the same problem with a smaller tolerance, 1.000E-08:

Lower limit of interval? 1

Upper limit of interval? 2

X value at t = 1.00000000E+00: 1

Number of values to return (1-500)? 10

Tolerance (> 0, default = 1.000E-06)? 1E-8

Direct output to one of the following:

(S)creen

(P)rinter

(F)ile

Lower limit: 1.00000000000000E+000

Upper limit: 2.00000000000000E+000

Value of X at 1.0000: 1.00000000000000E+000

Tolerance: 1.00000000000000E-008

t	X
1.00000000	1.00000000000000E+000
1.12208941	1.12982837391732E+000
1.20585321	1.22836146826667E+000
1.29271260	1.33921121906568E+000
1.38286653	1.46405185209736E+000
1.47648998	1.60468229863568E+000
1.57374241	1.76304147973215E+000
1.67477301	1.94122165006705E+000
1.77972398	2.14148082447423E+000
1.88873280	2.36625482837546E+000
2.00193373	2.61816928222327E+000

The exact solution is

$$X = t^2 - t \ln(t)$$

$$X(2) = 2.6137056$$

$$X(2.00193373) = 2.6181693$$

In the first run, a solution could be approximated within tolerance with a spacing of 0.1. In the second run, the algorithm had to vary the spacing in order to approximate a solution within the tolerance.

Solution to an Initial Value Problem for a First-Order Ordinary Differential Equation Using the Adams-Bashforth/Adams-Moulton Predictor/Corrector Scheme (ADAMS_1.INC)

Description

This example approximates the solution to a first-order ordinary differential equation with a specified initial condition using the four-step Adams-Bashforth/Adams-Moulton formulas (Burden and Faires 1985, 238–247). Runge-Kutta methods are one-step methods, because each calculation uses information from only one previous point. The Adams' formulas use information from four previous points, thus the four-step method.

Given a function of the form

$$dx/dt = TNTargetF(t, x)$$

which satisfies the conditions given at the beginning of this chapter, and an initial condition

$$x[LowerLimit] = XInitial$$

and spacing

$$h = (UpperLimit - LowerLimit)/NumIntervals$$

the fourth-order Runge-Kutta formula (see RUNGE_1.INC) is used to find approximations at the first three points in the interval $[LowerLimit, UpperLimit]$. Then the following explicit Adams-Bashforth formula:

$$\begin{aligned} x_o[i+1] = & x[i] + h/24 * \{ 55 * TNTargetF(t[i], x[i]) \\ & - 59 * TNTargetF(t[i-1], x[i-1]) \\ & + 37 * TNTargetF(t[i-2], x[i-2]) \\ & - 9 * TNTargetF(t[i-3], x[i-3]) \} \end{aligned}$$

and the following implicit Adams-Moulton formula:

$$\begin{aligned} x[i+1] = & x[i] + h/24 * \{ 9 * TNTargetF(t[i+1], x_o[i+1]) \\ & + 19 * TNTargetF(t[i], x[i]) \\ & - 5 * TNTargetF(t[i-1], x[i-1]) \\ & + TNTargetF(t[i-2], x[i-2]) \} \end{aligned}$$

approximate (predict) and refine (correct) all other points in the interval.

You must supply *UpperLimit*, *LowerLimit*, *XInitial*, *NumIntervals*, and *TNTargetF*.

User-Defined Types

TNvector = array[1..TNArraySize] of Real;

User-Defined Function

TNTargetF(t, X : Real) : Real;

$$dx/dt = \text{TNTargetF}(t, x)$$

Input Parameters

LowerLimit : Real; Lower limit of interval
UpperLimit : Real; Upper limit of interval
XInitial : Real; Value of *X* at *LowerLimit*
NumReturn : Integer; Number of (*t*, *x*) values to be returned from the procedure
NumIntervals : Integer; Number of subintervals to be used in calculations

The preceding parameters must satisfy the following conditions:

1. *NumReturn* > 0
2. *NumIntervals* ≥ *NumReturn*
3. *LowerLimit* ≠ *UpperLimit*

Output Parameters

TValues : TNvector; Values of *t* between the limits
XValues : TNvector; Values of *X* determined at the values in *TValues*
Error : Byte; 0: No errors
 1: *NumReturn* < 1
 2: *NumIntervals* < *NumReturn*
 3: *LowerLimit* = *UpperLimit*

Syntax of the Procedure Call

Adams(LowerLimit, UpperLimit, XInitial, NumReturn,
NumIntervals,TValues, XValues, Error);

The procedure *Adams* integrates the first-order differential equation *TNTargetF*.

Comments

This procedure will compute *NumIntervals* values in its calculations; however, you will rarely need to use the values. The vectors *TValues* and *XValues* will contain only *NumReturn* values at roughly equally spaced *t*-values between the lower and upper limits. (They will be equally spaced only when *NumIntervals* is a multiple of *NumReturn*.) Thus, you can ensure a highly accurate solution (by making *NumIntervals* large) without generating an excessive amount of output (by making *NumReturn* small).

The Adams-Bashforth/Adams-Moulton method uses the *New/Dispose* procedures to manipulate the heap and should not be used in any program that uses *Mark/Release* to manipulate the heap.

Warning: A stiff differential equation occurs when there are at least two very different scales of the independent variable on which the dependent variable(s) is changing; for example, $y = x + e^{-100x}$. The Adams-Bashforth/Adams-Moulton method may generate a numerical solution that bears no resemblance to the exact solution of the differential equation. This unstable numerical solution usually grows exponentially and may be oscillatory. However, if the exact solution of the differential equation grows as the independent variable increases, the instability may be difficult to detect. If a suspected instability has been encountered, reduce the interval size (*NumIntervals*).

Sample Program

The sample program ADAMS_1.PAS provides I/O functions that demonstrate the Adams-Bashforth/Adams-Moulton predictor/corrector method of solving initial value problems. Note that the file ADAMS_1.INC is included after the function *TNTargetF* is defined.

Example

Problem. Solve the following initial value problem with the Adams-Bashforth/Adams-Moulton method:

$$x' = x/t + t - 1 \quad 1 \leq t \leq 2$$
$$x(1) = 1$$

1. Code the differential equation into the program ADAMS_1.PAS:

```
function TNSolveF(t, X : Real) : Real;

(*****
****          THIS IS THE FIRST-ORDER DIFFERENTIAL EQUATION          ****
****          ****
*****
)

begin
  TNSolveF := x/t + t - 1;
end;                                { function TNSolveF }
```

2. Run ADAMS_1.PAS:

Lower limit of interval? 1

Upper limit of interval? 2

X value at t = 1.00000000E+00: 1

Number of values to return (1-500)? 10

Number of intervals (>= 10, default = 10)? 100

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

Lower limit: 1.00000000000000E+000
Upper limit: 2.00000000000000E+000
Value of X at 1.0000: 1.00000000000000E+000
Number of intervals: 100

t	X
1.00000000	1.00000000000000E+000
1.10000000	1.10515880229293E+000
1.20000000	1.22121413201736E+000
1.30000000	1.34892645643801E+000
1.40000000	1.48893886904034E+000
1.50000000	1.64180233820416E+000
1.60000000	1.80799419362396E+000
1.70000000	1.98793197365806E+000
1.80000000	2.18198400368348E+000
1.90000000	2.39047761682098E+000
2.00000000	2.61370563946811E+000

The exact solution is

$$X = t^2 - t \ln(t)$$
$$x(2) = 2.6137056$$

Solution to an Initial Value Problem for a Second-Order Ordinary Differential Equation Using the Runge-Kutta Method (RUNGE_2.INC)

Description

This example approximates the solution to a second-order ordinary differential equation with specified initial conditions using the two variable Runge-Kutta formulas (Burden and Faires 1985, 261–269).

Given a function of the form

$$d^2x/dt^2 = TNTargetF(t, x, x')$$

where x' indicates dx/dt (which satisfies the Lipschitz condition given at the beginning of this chapter), the initial conditions

$$x[LowerLimit] = InitialValue$$

$$x'[LowerLimit] = InitialDeriv$$

and spacing

$$h = (UpperLimit - LowerLimit)/NumIntervals$$

rewrite the second-order differential equation as two, first-order differential equations:

$$x' = y$$

$$y' = TNTargetF(t, x, y)$$

Then the fourth-order, two-variable Runge-Kutta method can be used to approximate simultaneously x and y (x and x').

The fourth-order Runge-Kutta formulas for these equations consist of the following:

$$F1x = h * y[t]$$

$$F1y = h * TNTargetF(t, x[t], y[t])$$

$$F2x = h * (y[t] + F1y/2)$$

$$F2y = h * TNTargetF(t + h/2, x[t] + F1x/2, y[t] + F1y/2)$$

$$F3x = h * (y[t] + F2y/2)$$

$$F3y = h * TNTargetF(t + h/2, x[t] + F2x/2, y[t] + F2y/2)$$

$$F4x = h * (y[t] + F3y)$$

$$F4y = h * TNTargetF(t + h, x[t] + F3x, y[t] + F3y)$$

$$x[t+1] = x[t] + (F1x + 2 * F2x + 2 * F3x + F4x)/6$$

$$y[t+1] = y[t] + (F1y + 2 * F2y + 2 * F3y + F4y)/6$$

where t ranges from *LowerLimit* to *UpperLimit* in steps of h . These formulas give a truncation error of order h^4 .

You must supply *LowerLimit*, *UpperLimit*, *XInitial*, *NumIntervals*, and *TNTargetF*.

User-Defined Types

TNvector = array[1..TNArraySize] of Real;

User-Defined Function

TNTargetF(t, X, XPrime : Real) : Real;

$$dx^2/dt^2 = \text{TNTargetF}(t, x, dx/dt)$$

Input Parameters

LowerLimit : Real;	Lower limit of interval
UpperLimit : Real;	Upper limit of interval
InitialValue : Real;	Value of X at <i>LowerLimit</i>
InitialDeriv : Real;	Derivative of X at <i>LowerLimit</i>
NumReturn : Integer;	Number of (t, x) values returned from the procedure
NumIntervals : Integer;	Number of subintervals used in the calculations

The preceding parameters must satisfy the following conditions:

1. $\text{NumReturn} > 0$
2. $\text{NumIntervals} \geq \text{NumReturn}$
3. $\text{LowerLimit} \neq \text{UpperLimit}$

Output Parameters

<code>TValues : TNvector;</code>	Values of t between the limits
<code>XValues : TNvector;</code>	Values of X determined at the values in <i>TValues</i>
<code>XDerivValues : TNvector;</code>	Values of the first derivative of X determined at the values in <i>TValues</i>
<code>Error : Byte;</code>	0: No errors 1: <i>NumReturn</i> < 1 2: <i>NumIntervals</i> < <i>NumReturn</i> 3: <i>LowerLimit</i> = <i>UpperLimit</i>

Syntax of the Procedure Call

```
InitialCond2ndOrder(LowerLimit, UpperLimit, InitialValue, InitialDeriv,  
                    NumReturn, NumIntervals, TValues, XValues,  
                    XDerivValues, Error);
```

The procedure *InitialCond2ndOrder* integrates the second-order differential equation *TNTargetF*.

Comments

This procedure will compute *NumIntervals* values in its calculations; however, you will rarely need to use all these values. The vectors *TValues*, *XValues*, and *XDerivValues* will contain only *NumReturn* values at roughly equally spaced t -values between the lower and upper limits. (They will be equally spaced only when *NumIntervals* is a multiple of *NumReturn*.) Thus, you can ensure a highly accurate solution (by making *NumIntervals* large) without generating an excessive amount of output (by making *NumReturn* small).

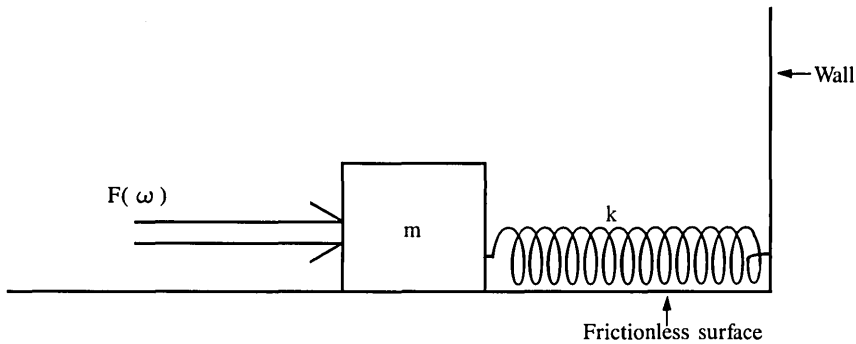
Warning: A differential equation occurs when there are at least two very different scales of the independent variable on which the dependent variable(s) is changing; for example, $y = x + e^{-100x}$. The Runge-Kutta method may generate a numerical solution that bears no resemblance to the exact solution of the differential equation. This unstable numerical solution usually grows exponentially and may be oscillatory. However, if the exact solution of the differential equation grows as the independent variable increases, the instability may be difficult to detect. If a suspected instability has been encountered, reduce the interval size (*NumIntervals*).

Sample Program

The sample program RUNGE_2.PAS provides I/O functions that demonstrate the Runge-Kutta method of solving initial value problems for second-order ordinary differential equations. Note that the file RUNGE_2.INC is included after the function *TNTargetF* is defined.

Example

Problem. A weight with mass m lies on a frictionless table and is connected to a spring with spring constant k :



If the weight is subject to a driving force $F \sin(\omega t)$ (ω represents the frequency of the driving force and t is time), the equation of motion of the mass is as follows:

$$m \frac{d^2 x}{dt^2} + kx = F \sin(\omega t)$$

Given

$$m = 2 \text{ kg}$$

$$F = 9 \text{ N}$$

$$k = 32 \text{ N/m}$$

$$\omega = 5 \text{ cycles/sec}$$

$$x(0) = 0 \text{ m}$$

$$dx(0)/dt = -2.5 \text{ m/sec}$$

find the position and velocity of the block from $t = 0$ second to $t = 2$ seconds.

1. Rewrite the preceding second-order differential equation:

$$\frac{d^2 x}{dt^2} = F/m \sin(\omega t) - k/m x$$

2. Code this second-order differential equation into the program RUNGE_2.PAS:

```
function TNTargetF(t : Real;
                  X : Real;
                  XPrime : Real) : Real;

(*****
****          THIS IS THE SECOND-ORDER DIFFERENTIAL EQUATION          ****
*****)

begin
  TNTargetF := 9/2 * Sin (5 * t) - 32/2 * x;
end; { function TNTargetF }
```

3. Run RUNGE_2.PAS:

Lower limit of interval? 0

Upper limit of interval? 2

Enter X value at t = 0.00000000E+00: 0

Enter derivative of X at t = 0.00000000E+00: -2.5

Number of values to return (1-500)? 10

Number of intervals (>= 10, default = 10)? 100

Direct output to one of the following:

(S)creen

(P)rinter

(F)ile

```
Lower limit: 0.00000000000000E+000
Upper limit: 2.00000000000000E+000
Value of X at 0.0000: 0.00000000000000E+000
Value of X' at 0.0000: -2.50000000000000E+000
Number of intervals: 100
```

t	Value of X	Derivative of X
0.00000000	0.00000000000000E+000	-2.50000000000000E+000
0.20000000	-4.20735284275848E-001	-1.35075642830665E+000
0.40000000	-4.54648724216734E-001	1.04036531118478E+000
0.60000000	-7.05605786993375E-002	2.47497991717220E+000
0.80000000	3.78400378699554E-001	1.63411037473655E+000
1.00000000	4.79461767300631E-001	-7.09151289407567E-001
1.20000000	1.39708469016311E-001	-2.40042152228323E+000
1.40000000	-3.28491796183335E-001	-1.88475529635974E+000
1.60000000	-4.94677974769030E-001	3.63745224811839E-001
1.80000000	-2.06059519715175E-001	2.27781864414105E+000
2.00000000	2.72008842396951E-001	2.09767516082021E+000

The exact solution is

$$x = \frac{F \sin(\omega t)}{m (\omega_o^2 - \omega^2)}$$

$$dx/dt = \frac{F \omega \cos(\omega t)}{m (\omega_o^2 - \omega^2)}$$

where ω_o is the natural frequency of the system

$$\omega_o^2 = k/m$$

The period of oscillation is given by

$$t = 2 \pi / \omega = 1.257 \text{ sec}$$

The data is taken from a function of which the derivative could be computed exactly. Following are the actual values:

t	Values of X	Derivative of X
0.0	0.000000000000E + 000	-2.500000000000E + 000
0.2	-4.207354924039E - 001	-1.350755764670E + 000
0.4	-4.546487134128E - 001	1.040367091367E + 000
0.6	-7.056000402993E - 002	2.474981241501E + 000
0.8	3.784012476539E - 001	1.634109052159E + 000
1.0	4.794621373315E - 001	-7.091554636580E - 001
1.2	1.397077490994E - 001	-2.400425716625E + 000
1.4	-3.284932993593E - 001	-1.884755635858E + 000
1.6	-4.946791233116E - 001	3.637500845215E - 001
1.8	-2.060592426208E - 001	2.277825654711E + 000
2.0	2.720105554446E - 001	2.097678822691E + 000

Solution to an Initial Value Problem for an n th-Order Ordinary Differential Equation Using the Runge-Kutta Method (RUNGE_N.INC)

Description

This example integrates an n th-order ordinary differential equation with specified initial conditions using the generalized Runge-Kutta formulas (Burden and Faires 1985, 261–269).

Given a function of the form

$$d^n x/dt^n = \text{TNTargetF}(t, x, x^{(1)}, \dots, x^{(n-1)})$$

where $x^{(j)}$ indicates $d^j x/dt^j$, which satisfies the general Lipschitz condition (the Lipschitz condition for first-order and second-order ordinary differential equations is given at the beginning of this chapter, and initial condition

$$x[\text{LowerLimit}] = a_1$$

$$x^{(1)}[\text{LowerLimit}] = a_2$$

⋮

$$x^{(n-1)}[\text{LowerLimit}] = a_n$$

and spacing

$$h = (\text{UpperLimit} - \text{LowerLimit})/\text{NumIntervals}$$

rewrite the n th-order differential equation as n first-order differential equations:

$$x^{(1)} = y_1$$

$$x^{(2)} = y_1^{(1)} = y_2$$

$$x^{(3)} = y_2^{(1)} = y_3$$

⋮

$$x^{(n-1)} = y_{n-2}^{(1)} = y_{n-1}$$

$$x^{(n)} = y_{n-1}^{(1)} = \text{TNTargetF}(t, x, y_1, y_2, \dots, y_{n-1})$$

Then the fourth-order general Runge-Kutta method can be used to approximate simultaneously the y 's (x and its derivatives).

The general Runge-Kutta formulas for these equations consist of the following:

$$F1x = h * y_1[t]$$

$$F1y_1 = h * y_2[t]$$

.

$$F1y_{n-2} = h * y_{n-1}[t]$$

$$F1y_{n-1} = h * TNTargetF(t, x[t], y_1[t], ..., y_{n-1}[t])$$

$$F2x = h * (y_1[t] + F1y_1/2)$$

$$F2y_1 = h * (y_2[t] + F1y_2/2)$$

.

$$F2y_{n-2} = h * (y_{n-1}[t] + F1y_{n-1}/2)$$

$$F2y_{n-1} = h * TNTargetF(t + h/2, x[t] + F1x/2, y_1[t] + F1y_1/2, ..., y_{n-1}[t] + F1y_{n-1}/2)$$

$$F3x = h * (y_1[t] + F2y_1/2)$$

$$F3y_1 = h * (y_2[t] + F2y_2/2)$$

.

$$F3y_{n-2} = h * (y_{n-1}[t] + F2y_{n-1}/2)$$

$$F3y_{n-1} = h * TNTargetF(t + h/2, x[t] + F2x/2, y_1[t] + F2y_1/2, ..., y_{n-1}[t] + F2y_{n-1}/2)$$

$$F4x = h * (y_1[t] + F3y_1)$$

$$F4y_1 = h * (y_2[t] + F3y_2)$$

.

$$F4y_{n-2} = h * (y_{n-1}[t] + F3y_{n-1})$$

$$F4y_{n-1} = h * TNTargetF(t + h, x[t] + F3x, y_1[t] + F3y_1, ..., y_{n-1}[t] + F3y_{n-1})$$

$$x[t+1] = x[t] + (F1x + 2 * F2x + 2 * F3x + F4x)/6$$

$$y_1[t+1] = y_1[t] + (F1y_1 + 2 * F2y_1 + 2 * F3y_1 + F4y_1)/6$$

$$y_2[t+1] = y_2[t] + (F1y_2 + 2 * F2y_2 + 2 * F3y_2 + F4y_2)/6$$

.

$$y_{n-2}[t+1] = y_{n-2}[t] + (F1y_{n-2} + 2 * F2y_{n-2} + 2 * F3y_{n-2} + F4y_{n-2})/6$$

$$y_{n-1}[t+1] = y_{n-1}[t] + (F1y_{n-1} + 2 * F2y_{n-1} + 2 * F3y_{n-1} + F4y_{n-1})/6$$

where t ranges from *LowerLimit* to *UpperLimit* in steps of h . These formulas give a truncation error of order h^4 .

You must supply the order, limits, initial values, and *TNTargetF*. The order may be arbitrarily large.

User-Defined Types

```
TNvector = array[0..TNRowSize] of Real;
TNmatrix = array[0..TNColumnSize] of TNvector;
```

TNRowSize is an upper bound for the number of values returned for a particular variable (*NumReturn*). *TNColumnSize* is an upper bound for the order of the differential equation (*Order*).

User-Defined Function

```
TNTargetF(V : TNvector) : Real;
```

The elements of *V* are defined as

V[0] corresponds to *t*
V[1] corresponds to *x*
V[2] corresponds to first derivative of *x*
V[3] corresponds to second derivative of *x*
 .
 .
 .

This is the differential equation:

$d^n x/dt^n = \text{TNTargetF}(t, x, x^{(1)}, \dots, x^{(n-1)})$ where *n* is the order of the equation.

The procedure *InitialCondition* integrates this *n*th-order differential equation.

Input Parameters

Order : Integer;	Order of the differential equation
LowerLimit : Real;	Lower limit of interval
UpperLimit : Real;	Upper limit of interval
InitialValues : TNvector;	Values of <i>X</i> and its derivatives at <i>LowerLimit</i>
NumReturn : Integer;	Number of $(t, x, x^{(1)}, \dots, x^{(n)})$ values returned from the procedure
NumIntervals : Integer;	Number of subintervals used in the calculations

The preceding parameters must satisfy the following conditions:

1. $NumReturn > 0$
2. $NumIntervals \geq NumReturn$
3. $Order > 0$
4. $LowerLimit \neq UpperLimit$

Output Parameters

`SolutionValues` : TNmatrix; Values of t , x and the derivatives of x between the limits

`Error` : Byte;

0: No errors

1: $NumReturn < 1$

2: $NumIntervals < NumReturn$

3: $Order < 1$

4: $LowerLimit = UpperLimit$

Syntax of the Procedure Call

`InitialCondition(Order, LowerLimit, UpperLimit, InitialValues,
NumReturn, NumIntervals, SolutionValues, Error);`

Comments

The first row of *SolutionValues* will be the values of t between the limits, the second row of *SolutionValues* will be the values of x between the limits, the third row of *SolutionValues* will be the values of $x^{(1)}$ between the limits, and so on.

This procedure will compute *NumIntervals* values in its calculations; however, you will rarely need to use all those values. The rows of *SolutionValues* will contain only *NumReturn* values at roughly equally spaced t -values between the lower and upper limits. (They will be equally spaced only when *NumIntervals* is a multiple of *NumReturn*.) Thus, you can ensure a highly accurate solution (by making *NumIntervals* large) without generating an excessive amount of output (by making *NumReturn* small).

There are no bounds on the order of the differential equation.

This routine stores much information on the heap. If you try to solve a high-order differential equation very precisely (that is, both *Order* and *NumIntervals* are large), you may get run-time error \$FF, Heap/Stack collision. If this happens, the dimension of *TNvector* and *TNmatrix* should be reduced as much as possible. If this is not possible, remove any RAM-resident software (for example, SideKick, SuperKey, or a print buffer).

The Runge-Kutta method uses the *New/Dispose* procedures to manipulate the heap and should not be used in any program that uses *Mark/Release* to manipulate the heap.

Warning: A stiff differential equation occurs when there are at least two very different scales of the independent variable on which the dependent variable(s) is changing; for example, $y = x + e^{-100x}$. The Runge-Kutta method may generate a numerical solution that bears no resemblance to the exact solution of the differential equation. This unstable numerical solution usually grows exponentially and may be oscillatory. However, if the exact solution of the differential equation grows as the independent variable increases, the instability may be difficult to detect. If a suspected instability has been encountered, reduce the interval size (*NumIntervals*).

Sample Program

The sample program RUNGE_N.PAS provides I/O functions that demonstrate the Runge-Kutta method of solving initial value problems for high-order ordinary differential equations. Note that the file RUNGE_N.INC is included after the function *TNTargetF* is defined.

Example

Problem. Find the solution to the following fourth-order ordinary differential equation from $t = 0$ to $t = 1$:

$$d^4x(t)/dt^4 = -4x(t) d^3x(t)/dt^3$$

$$x(0) = 1$$

$$dx(0)/dt = -1$$

$$d^2x(0)/dt^2 = 2$$

$$d^3x(0)/dt^3 = -6$$

1. Code the equation into the program RUNGE_N.PAS:

```
function TNTargetF(V : TNvector) : Real;

(*****
(*   THIS IS THE DIFFERENTIAL EQUATION   *)
(*****
(*
(*   dn x           (1)           (n-1)   *)
(*   ----- = TNTargetF(t, x, x(1), ... x(n-1)) *)
(*   dtn                                     *)
(*
(* where n is the order of the equation. *)
(*
(* The elements of V are defined: *)
(* V[0] corresponds to t *)
(* V[1] corresponds to X *)
(* V[2] corresponds to 1st derivative of X *)
(* V[3] corresponds to 2nd derivative of X *)
(* . *)
(* . *)
(* . *)
(*****)
```

```
begin
  TNTargetF := -4 * V[1] * V[4];
end; { function TNTargetF }
```

2. Run RUNGE_N.PAS:

Order of the equation (1-10)? 4

Lower limit of interval? 0

Upper limit of interval? 1

Enter X value at t = 0.000000E+000: 1
 Derivative 1 of X at t = 0.000000E+000: -1
 Derivative 2 of X at t = 0.000000E+000: 2
 Derivative 3 of X at t = 0.000000E+000: -6

Number of values to return (1-100)? 10

Number of intervals (>= 10, default = 10)? 100

Direct output to one of the following:

(S)creen
 (P)rinter
 (F)ile

Lower limit: 0.000000000000000E+000
 Upper limit: 1.000000000000000E+000
 Number of intervals: 100

Initial conditions at lower limit:

X[1]= 1.00000000000000E+000
 X[2]= -1.00000000000000E+000
 X[3]= 2.00000000000000E+000
 X[4]= -6.00000000000000E+000

t	Value X[1]
0.00000000	1.00000000000000E+000
0.10000000	9.09090909737517E-001
0.20000000	8.33333334189337E-001
0.30000000	7.69230770157394E-001
0.40000000	7.14285715280102E-001
0.50000000	6.66666667788519E-001
0.60000000	6.25000001337168E-001
0.70000000	5.88235295769619E-001
0.80000000	5.55555557625526E-001
0.90000000	5.26315792064849E-001
1.00000000	5.00000003213983E-001

t	Value X[2]
0.00000000	-1.00000000000000E+000
0.10000000	-8.26446283273189E-001
0.20000000	-6.94444446826215E-001
0.30000000	-5.91715977923112E-001
0.40000000	-5.10204082090465E-001
0.50000000	-4.44444443661452E-001
0.60000000	-3.90624997971428E-001
0.70000000	-3.46020758007957E-001
0.80000000	-3.08641970911504E-001
0.90000000	-2.77008304743045E-001
1.00000000	-2.49999993429933E-001

t	Value X[3]
0.00000000	2.00000000000000E+000
0.10000000	1.50262961438149E+000
0.20000000	1.15740742373768E+000
0.30000000	9.10332288053840E-001
0.40000000	7.28862989793594E-001
0.50000000	5.92592607536865E-001
0.60000000	4.88281263842229E-001
0.70000000	4.07083261374878E-001
0.80000000	3.42935540127152E-001
0.90000000	2.91587706310718E-001
1.00000000	2.50000010753535E-001

t	Value X[4]
0.00000000	-6.00000000000000E+000
0.10000000	-4.09808076056272E+000
0.20000000	-2.89351855059016E+000
0.30000000	-2.10076680857258E+000
0.40000000	-1.56184925333600E+000
0.50000000	-1.18518520443061E+000
0.60000000	-9.15527359078898E-001
0.70000000	-7.18382215400418E-001
0.80000000	-5.71559223064178E-001
0.90000000	-4.60401631119694E-001
1.00000000	-3.75000005740567E-001

$X[1]$ are the values of $x(t)$.

$X[2]$ are the values of $dx(t)/dt$.

$X[3]$ are the values of $d^2x(t)/dt^2$.

$X[4]$ are the values of $d^3x(t)/dt^3$.

The exact solution is

$$\begin{aligned}x(t) &= (t+1)^{-1} \\dx(t)/dt &= -(t+1)^{-2} \\d^2x(t)/dt^2 &= 2(t+1)^{-3} \\d^3x(t)/dt^3 &= -6(t+1)^{-4}\end{aligned}$$

$$\begin{aligned}x(1) &= 0.5 \\dx(1)/dt &= -0.25 \\d^2x(1)/dt^2 &= 0.25 \\d^3x(1)/dt^3 &= -0.375\end{aligned}$$

Solution to an Initial Value Problem for a System of Coupled First-Order Ordinary Differential Equations Using the Runge-Kutta Method (RUNGE_S1.INC)

Description

This example integrates a system of coupled first-order ordinary differential equations with specified initial conditions using the generalized Runge-Kutta formulas (Burden and Faires 1985, 261–269).

Given m first-order ordinary differential equations in the form

$$dx_1/dt = \text{TNTargetF1}(t, x_1, x_2, \dots, x_m)$$

$$dx_2/dt = \text{TNTargetF2}(t, x_1, x_2, \dots, x_m)$$

.

.

.

$$dx_m/dt = \text{TNTargetFm}(t, x_1, x_2, \dots, x_m)$$

which satisfies the Lipshitz condition (the Lipshitz condition for first-order and second-order ordinary differential equations is given at the beginning of this chapter; consult the previous book reference for details of the Lipshitz condition for systems), and initial conditions

$$x_1[\text{LowerLimit}] = a_1$$

$$x_2[\text{LowerLimit}] = a_2$$

.

.

.

$$x_m[\text{LowerLimit}] = a_m$$

and spacing

$$h = (\text{UpperLimit} - \text{LowerLimit})/\text{NumIntervals}$$

the fourth-order general Runge-Kutta method can be used to approximate simultaneously the x_j 's.

The general Runge-Kutta formulas for these equations are as follows:

$$\begin{aligned}
F1x_1 &= h * TNTargetF1(t, x_1[t], x_2[t], ..., x_m[t]) \\
F1x_2 &= h * TNTargetF2(t, x_1[t], x_2[t], ..., x_m[t]) \\
&\vdots \\
F1x_m &= h * TNTargetFm(t, x_1[t], x_2[t], ..., x_m[t]) \\
F2x_1 &= h * TNTargetF1(t + h/2, x_1[t] + F1x_1/2, x_2[t] + F1x_2/2, ..., x_m[t] \\
&\quad + F1x_m/2) \\
F2x_2 &= h * TNTargetF2(t + h/2, x_1[t] + F1x_1/2, x_2[t] + F1x_2/2, ..., x_m[t] \\
&\quad + F1x_m/2) \\
&\vdots \\
F2x_m &= h * TNTargetFm(t + h/2, x_1[t] + F1x_1/2, x_2[t] + F1x_2/2, ..., x_m[t] \\
&\quad + F1x_m/2) \\
F3x_1 &= h * TNTargetF1(t + h/2, x_1[t] + F2x_1/2, x_2[t] + F2x_2/2, ..., x_m[t] \\
&\quad + F2x_m/2) \\
F3x_2 &= h * TNTargetF2(t + h/2, x_1[t] + F2x_1/2, x_2[t] + F2x_2/2, ..., x_m[t] \\
&\quad + F2x_m/2) \\
&\vdots \\
F3x_m &= h * TNTargetFm(t + h/2, x_1[t] + F2x_1/2, x_2[t] + F2x_2/2, ..., x_m[t] \\
&\quad + F2x_m/2) \\
F4x_1 &= h * TNTargetF1(t + h, x_1[t] + F3x_1, x_2[t] + F3x_2, ..., x_m[t] + F3x_m) \\
F4x_2 &= h * TNTargetF2(t + h, x_1[t] + F3x_1, x_2[t] + F3x_2, ..., x_m[t] + F3x_m) \\
&\vdots \\
F4x_m &= h * TNTargetFm(t + h, x_1[t] + F3x_1, x_2[t] + F3x_2, ..., x_m[t] + F3x_m) \\
x_1[t+1] &= x_1[t] + (F1x_1 + 2*F2x_1 + 2*F3x_1 + F4x_1)/6 \\
x_2[t+1] &= x_2[t] + (F1x_2 + 2*F2x_2 + 2*F3x_2 + F4x_2)/6 \\
&\vdots \\
x_m[t+1] &= x_m[t] + (F1x_m + 2*F2x_m + 2*F3x_m + F4x_m)/6
\end{aligned}$$

where t ranges from *LowerLimit* to *UpperLimit* in steps of h . These formulas give a truncation error of order h^4 .

You must supply the number of differential equations, the limits, initial values, and *TNTargetF*'s.

This procedure can solve a system of up to ten differential equations (see "Comments" for information about how to increase this limit).

User-Defined Types

```
TNvector = array[0..TNRowSize] of Real;
TNmatrix = array[0..TNColumnSize] of TNvector;
```

TNRowSize is an upper bound for the number of values returned for a particular variable (*NumReturn*). *TNColumnSize* is an upper bound for the number of differential equations (*NumEquations*).

User-Defined Functions

```
function TNTargetF1(V : TNvector) : Real;
function TNTargetF2(V : TNvector) : Real;
function TNTargetF3(V : TNvector) : Real;
function TNTargetF4(V : TNvector) : Real;
function TNTargetF5(V : TNvector) : Real;
function TNTargetF6(V : TNvector) : Real;
function TNTargetF7(V : TNvector) : Real;
function TNTargetF8(V : TNvector) : Real;
function TNTargetF9(V : TNvector) : Real;
function TNTargetF10(V : TNvector) : Real;
```

These are the differential equations:

$$dx_j/dt = TNTargetF_j(t, x_1, x_2, \dots, x_{10})$$

where j ranges from 1 to 10.

The elements of the vector V are defined as follows:

$$\begin{aligned} V[0] &= t \\ V[1] &= x_1 \\ V[2] &= x_2 \\ &\vdots \\ V[10] &= x_{10} \end{aligned}$$

The procedure defined in RUNGE_S1.INC solves this system of coupled differential equations (a maximum of ten equations). All ten functions must be defined, even if your system contains less than ten equations.

Input Parameters

NumEquations : Integer; Number of first-order differential equations
 LowerLimit : Real; Lower limit of interval
 UpperLimit : Real; Upper limit of interval
 InitialValues : TNvector; Values of x_1, x_2, \dots, x_m at *LowerLimit*
 NumReturn : Integer; Number of $(t, x_1, x_2, \dots, x_m)$ values returned from the procedure
 NumIntervals : Integer; Number of subintervals used in the calculations

The preceding parameters must satisfy the following conditions:

1. $NumReturn > 0$
2. $NumIntervals \geq NumReturn$
3. $NumEquations > 0$
4. $LowerLimit \neq UpperLimit$

Output Parameters

SolutionValues : TNmatrix; Values of t, x_1, x_2, \dots, x_m between the limits
 Error : Byte; 0: No errors
 1: $NumReturn < 1$
 2: $NumIntervals < NumReturn$
 3: $NumEquations < 1$
 4: $LowerLimit = UpperLimit$

Syntax of the Procedure Call

```
InitialConditionSystem(NumEquations, LowerLimit, UpperLimit,  
                      InitialValues, NumReturn, NumIntervals,  
                      SolutionValues, Error);
```

Comments

The first row of *SolutionValues* will be the values of t between the limits, the second row of *SolutionValues* will be the values of x_1 between the limits, the third row of *SolutionValues* will be the values of x_2 between the limits, and so on.

All ten user-defined functions are called from the procedure. If your system has less than ten equations, you must still define all ten functions or the program will not compile. The superfluous functions should be defined as follows (*TNTargetF10* is used as an example):

```
function TNTargetF10(V : TNvector) : Real;  
  
begin  
end;           { function TNTargetF10 }
```

If you need to solve a system with more than ten equations, then edit the include file *RUNGE_S1.INC*. The following line should be added to the end of procedure *Step*:

```
F[11] := Spacing * TNTargetF11(CurrentValues);
```

More statements (for $F[12]$, and so on) may be added as necessary. All new functions (for example, *TNTargetF11*) must be defined in your top-level program. **Note:** Before making any changes to the include file, make sure you have a backup copy.

This procedure will compute *NumIntervals* values in its calculations; however, you will rarely need to use these values. The rows of *SolutionValues* will contain only *NumReturn* values at roughly equally spaced t -values between the lower and upper limits. (They will be equally spaced only when *NumIntervals* is a multiple of *NumReturn*.) Thus, you can ensure a highly accurate solution (by making *NumIntervals* large) without generating an excessive amount of output (by making *NumReturn* small).

This routine stores much information on the heap. If you try to accurately solve a large system (that is, if both *NumEquations* and *NumIntervals* are large), you may get run-time error \$FF, Heap/Stack collision. If this happens, the dimension of *TNvector* and *TNmatrix* should be reduced as much as possible. If this is not possible, then remove any RAM-resident software (for example, SideKick, SuperKey, or a print buffer).

The Runge-Kutta method uses the *New/Dispose* procedures to manipulate the heap and should not be used in any program that uses *Mark/Release* to manipulate the heap.

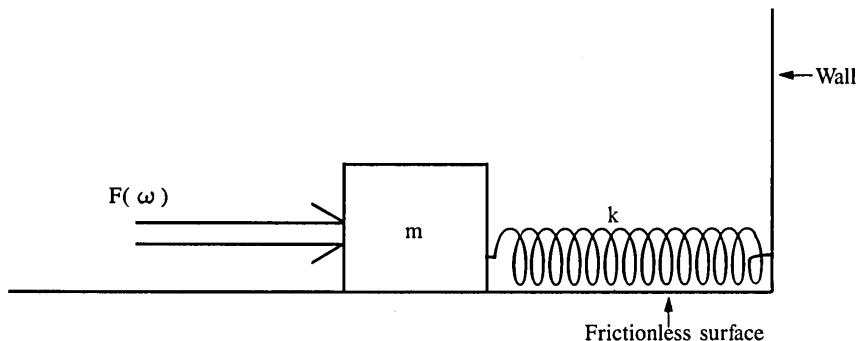
Warning: A stiff differential equation occurs when there are at least two very different scales of the independent variable on which the dependent variable(s) is changing; for example, $y = x + e^{-100x}$. The Runge-Kutta method may generate a numerical solution that bears no resemblance to the exact solution of the differential equation. This unstable numerical solution usually grows exponentially and may be oscillatory. However, if the exact solution of the differential equation grows as the independent variable increases, the instability may be difficult to detect. If a suspected instability has been encountered, reduce the interval size (*NumIntervals*).

Sample Program

The sample program RUNGE_S1.PAS provides I/O functions that demonstrate the Runge-Kutta method of solving initial value problems for systems of first-order ordinary differential equations. Note that the file RUNGE_S1.INC is included after the *TNTargetF* functions are defined.

Example

Problem. A weight with mass m lays on a frictionless table and is connected to a spring with spring constant k :



If the mass is subject to a driving force $F \sin(\omega t)$ (ω represents the frequency of the driving force and t is time), the equation of motion of the mass is as follows:

$$m \frac{d^2 x}{dt^2} + k x = F \sin(\omega t)$$

Given

$$m = 2 \text{ kg}$$

$$F = 9 \text{ N}$$

$$k = 32 \text{ N/m}$$

$$\omega = 5 \text{ cycles/sec}$$

$$x(0) = 0 \text{ m}$$

$$dx(0)/dt = -2.5 \text{ m/sec}$$

find the position and velocity of the block from $t = 0$ second to $t = 2$ seconds.

1. Write the second-order ordinary differential equations as a system of two coupled first-order ordinary differential equations:

$$dx_1/dt = x_2$$

$$dx_2/dt = (F/m) \sin(\omega t) - (k/m) x_1$$

2. Code these equations into the program RUNGE_S1.PAS:

```
function TNTargetF1(V : TNvector) : Real;

(*****
* THIS IS THE FIRST DIFFERENTIAL EQUATION
*****
*)
(*
* dx[1]
* ----- = TNTargetF1(t, x[1], x[2], ... x[m])
* dt
*)
(* The vector V is defined:
*)
(* V[0] = t
*)
(* V[1] = X[1]
*)
(* V[2] = X[2]
*)
(* .
*)
(* .
*)
(* .
*)
(* V[m] = X[m]
*)
(*
*)
(* where m is the number of coupled equations.
*)
(*****)

begin
  TNTargetF1 := V[2];
end;
{ function TNTargetF1 }

function TNTargetF2(V : TNvector) : Real;
```

```

(*****)
(* THIS IS THE SECOND DIFFERENTIAL EQUATION *)
(*****)
(* *)
(* dx[2] *)
(* ----- = TNTargetF2(t, x[1], x[2], ... x[m]) *)
(* dt *)
(* *)
(* The vector V is defined: *)
(* V[0] = t *)
(* V[1] = X[1] *)
(* V[2] = X[2] *)
(* . *)
(* . *)
(* . *)
(* V[m] = X[m] *)
(* *)
(* where m is the number of coupled equations. *)
(*****)

```

```

begin
  TNTargetF2 := 9/2 * Sin(5 * V[0]) - 32/2 * V[1];
end; { function TNTargetF2 }

```

```

function TNTargetF3(V : TNvector) : Real;

```

```

(*****)
(* THIS IS THE THIRD DIFFERENTIAL EQUATION *)
(*****)
(* *)
(* dx[3] *)
(* ----- = TNTargetF3(t, x[1], x[2], ... x[m]) *)
(* dt *)
(* *)
(* The vector V is defined: *)
(* V[0] = t *)
(* V[1] = X[1] *)
(* V[2] = X[2] *)
(* . *)
(* . *)
(* . *)
(* V[m] = X[m] *)
(* *)
(* where m is the number of coupled equations. *)
(*****)

```

```

begin
end; { function TNTargetF3 }

```

Functions *TNTarget4* to *TNTarget10* should be defined like the function *TNTargetF3*.

3. Run RUNGE_S1.PAS:

Number of first order equations: (1-10)? 2

Lower limit of interval? 0

Upper limit of interval? 2

Enter X[1] value at t = 0.00000000E+00: 0

Enter X[2] value at t = 0.00000000E+00: -2.5

Number of values to return (1-100)? 10

Number of intervals (> = 10, default=10)? 100

Direct output to one of the following:

(S)creen

(P)rinter

(F)ile

Lower limit: 0.00000000000000E+000

Upper limit: 2.00000000000000E+000

Number of intervals: 100

Initial conditions at lower limit:

X[1] = 0.00000000000000E+000

X[2] = -2.50000000000000E+000

t	Value X[1]
0.00000000	0.00000000000000E+000
0.20000000	-4.20735284275848E-001
0.40000000	-4.54648724216734E-001
0.60000000	-7.05605786993375E-002
0.80000000	3.78400378699554E-001
1.00000000	4.79461767300631E-001
1.20000000	1.39708469016311E-001
1.40000000	-3.28491796183335E-001
1.60000000	-4.94677974769030E-001
1.80000000	-2.06059519715175E-001
2.00000000	2.72008842396951E-001

t	Value X[2]
0.00000000	-2.50000000000000E+000
0.20000000	-1.35075642830665E+000
0.40000000	1.04036531118478E+000
0.60000000	2.47497991717220E+000
0.80000000	1.63411037473655E+000
1.00000000	-7.09151289407567E-001
1.20000000	-2.40042152228323E+000
1.40000000	-1.88475529635974E+000
1.60000000	3.63745224811839E-001
1.80000000	2.27781864414105E+000
2.00000000	2.09767516082021E+000

X[1] are the values of $x(t)$, the position. X[2] are the values of $dx(t)/dt$, the velocity.

The exact solution is

$$x = \frac{F \sin(\omega t)}{m (\omega_o^2 - \omega^2)}$$

$$dx/dt = \frac{F \omega \cos(\omega t)}{m (\omega_o^2 - \omega^2)}$$

where ω_o is the natural frequency of the system:

$$\omega_o^2 = k/m$$

The period of oscillation is given by

$$T = 2 \pi / \omega = 1.257 \text{ sec}$$

The data is taken from a function of which the derivative could be computed exactly. The actual values are as follows:

t	Values of X	Derivative of X
0.0	0.000000000000E + 000	-2.500000000000E + 000
0.2	-4.207354924039E - 001	-1.350755764670E + 000
0.4	-4.546487134128E - 001	1.040367091367E + 000
0.6	-7.056000402993E - 002	2.474981241501E + 000
0.8	3.784012476539E - 001	1.634109052159E + 000
1.0	4.794621373315E - 001	-7.091554636580E - 001
1.2	1.397077490994E - 001	-2.400425716625E + 000
1.4	-3.284932993593E - 001	-1.884755635858E + 000
1.6	-4.946791233116E - 001	3.637500845215E - 001
1.8	-2.060592426208E - 001	2.277825654711E + 000
2.0	2.720105554446E - 001	2.097678822691E + 000

Solution to an Initial Value Problem for a System of Coupled Second-Order Ordinary Differential Equations Using the Runge-Kutta Method (RUNGE_S2.INC)

Description

This example integrates a system of coupled second-order ordinary differential equations with specified initial conditions using the generalized Runge-Kutta formulas (Burden and Faires 1985, 261–269).

Given m coupled second-order ordinary differential equations of the form

$$d^2x_1/dt^2 = TNTargetF1(t, x_1, x'_1, x_2, x'_2, \dots, x_m, x'_m)$$

$$d^2x_2/dt^2 = TNTargetF2(t, x_1, x'_1, x_2, x'_2, \dots, x_m, x'_m)$$

.

$$d^2x_m/dt^2 = TNTargetFm(t, x_1, x'_1, x_2, x'_2, \dots, x_m, x'_m)$$

where x'_j indicates dx_j/dt , which satisfies the Lipshitz condition (the Lipshitz condition for first-order and second-order ordinary differential equations is given at the beginning of this chapter; consult the previous book reference for details of the Lipshitz condition for systems), and initial condition

$$x_1[LowerLimit] = a_1 \qquad x'_1[LowerLimit] = b_1$$

$$x_2[LowerLimit] = a_2 \qquad x'_2[LowerLimit] = b_2$$

.

$$x_m[LowerLimit] = a_m \qquad x'_m[LowerLimit] = b_m$$

and spacing

$$h = (UpperLimit - LowerLimit)/NumIntervals$$

rewrite each of the second-order differential equations as two, first-order differential equations:

$$\begin{aligned}
 dx_1/dt &= y_1 \\
 dy_1/dt &= TNTargetF1(t, x_1, y_1, x_2, y_2, \dots, x_m, y_m) \\
 dx_2/dt &= y_2 \\
 dx_2/dt &= TNTargetF2(t, x_1, y_1, x_2, y_2, \dots, x_m, y_m) \\
 &\vdots \\
 &\vdots \\
 dx_m/dt &= y_m \\
 dx_m/dt &= TNTargetFm(t, x_1, y_1, x_2, y_2, \dots, x_m, y_m)
 \end{aligned}$$

Then the fourth-order general Runge-Kutta method can be used to approximate the x_i 's and the y_i 's simultaneously.

The general Runge-Kutta formulas for these equations are as follows:

$$\begin{aligned}
 Flx_1 &= h * y_1 \\
 Fly_1 &= h * TNTargetF1(t, x_1[t], y_1[t], x_2[t], y_2[t], \dots, x_m[t], y_m[t]) \\
 Flx_2 &= h * y_2 \\
 Fly_2 &= h * TNTargetF2(t, x_1[t], y_1[t], x_2[t], y_2[t], \dots, x_m[t], y_m[t]) \\
 &\vdots \\
 &\vdots \\
 Flx_m &= h * y_m \\
 Fly_m &= h * TNTargetFm(t, x_1[t], y_1[t], x_2[t], y_2[t], \dots, x_m[t], y_m[t]) \\
 \\
 F2x_1 &= h * (y_1 + Fly_1/2) \\
 F2y_1 &= h * TNTargetF1(t + h/2, x_1[t] + Flx_1/2, y_1[t] + Fly_1/2, x_2[t] \\
 &\quad + Flx_2/2, y_2[t] + Fly_2/2, \dots, x_m[t] + Flx_m/2, y_m[t] + Fly_m/2) \\
 F2x_2 &= h * (y_2 + Fly_2/2) \\
 F2y_2 &= h * TNTargetF2(t + h/2, x_1[t] + Flx_1/2, y_1[t] + Fly_1/2, x_2[t] \\
 &\quad + Flx_2/2, y_2[t] + Fly_2/2, \dots, x_m[t] + Flx_m/2, y_m[t] + Fly_m/2) \\
 &\vdots \\
 &\vdots \\
 F2x_m &= h * (y_m + Fly_m/2) \\
 F2y_m &= h * TNTargetFm(t + h/2, x_1[t] + Flx_1/2, y_1[t] + Fly_1/2, x_2[t] \\
 &\quad + Flx_2/2, y_2[t] + Fly_2/2, \dots, x_m[t] + Flx_m/2, y_m[t] + Fly_m/2)
 \end{aligned}$$

$$F3x_1 = h * (y_1 + F2y_1/2)$$

$$F3y_1 = h * TNTargetF1(t + h/2, x_1[t] + F2x_1/2, y_1[t] + F2y_1/2, x_2[t] + F2x_2/2, y_2[t] + F2y_2/2, ..., x_m[t] + F2x_m/2, y_m[t] + F2y_m/2)$$

$$F3x_2 = h * (y_2 + F2y_2/2)$$

$$F3y_2 = h * TNTargetF2(t + h/2, x_1[t] + F2x_1/2, y_1[t] + F2y_1/2, x_2[t] + F2x_2/2, y_2[t] + F2y_2/2, ..., x_m[t] + F2x_m/2, y_m[t] + F2y_m/2)$$

⋮

$$F3x_m = h * (y_m + F2y_m/2)$$

$$F3y_m = h * TNTargetFm(t + h/2, x_1[t] + F2x_1/2, y_1[t] + F2y_1/2, x_2[t] + F2x_2/2, y_2[t] + F2y_2/2, ..., x_m[t] + F2x_m/2, y_m[t] + F2y_m/2)$$

$$F4x_1 = h * (y_1 + F3y_1)$$

$$F4y_1 = h * TNTargetF1(t + h, x_1[t] + F3x_1, y_1[t] + F3y_1, x_2[t] + F3x_2, y_2[t] + F3y_2, ..., x_m[t] + F3x_m, y_m[t] + F3y_m)$$

$$F4x_2 = h * (y_2 + F3y_2)$$

$$F4y_2 = h * TNTargetF2(t + h, x_1[t] + F3x_1, y_1[t] + F3y_1, x_2[t] + F3x_2, y_2[t] + F3y_2, ..., x_m[t] + F3x_m, y_m[t] + F3y_m)$$

⋮

$$F4x_m = h * (y_m + F3y_m)$$

$$F4y_m = h * TNTargetFm(t + h, x_1[t] + F3x_1, y_1[t] + F3y_1, x_2[t] + F3x_2, y_2[t] + F3y_2, ..., x_m[t] + F3x_m, y_m[t] + F3y_m)$$

$$x_1[t+1] = x_1[t] + (F1x_1 + 2 * F2x_1 + 2 * F3x_1 + F4x_1)/6$$

$$y_1[t+1] = y_1[t] + (F1y_1 + 2 * F2y_1 + 2 * F3y_1 + F4y_1)/6$$

$$x_2[t+1] = x_2[t] + (F1x_2 + 2 * F2x_2 + 2 * F3x_2 + F4x_2)/6$$

$$y_2[t+1] = y_2[t] + (F1y_2 + 2 * F2y_2 + 2 * F3y_2 + F4y_2)/6$$

⋮

$$x_m[t+1] = x_m[t] + (F1x_m + 2 * F2x_m + 2 * F3x_m + F4x_m)/6$$

$$y_m[t+1] = y_m[t] + (F1y_m + 2 * F2y_m + 2 * F3y_m + F4y_m)/6$$

where t ranges from *LowerLimit* to *UpperLimit* in steps of h . These formulas give a truncation error of order h^4 .

You must supply the number of equations, limits, initial values, and *TNTargetF*'s.

This procedure can solve a system of up to ten, second-order ordinary differential equations (see "Comments" for information about how to increase this limit).

User-Defined Types

```
TNData = record
    x : Real;
    xDeriv : Real;
end; { TNData record }
TNvector = array[0..TNRowSize] of TNData;
TNmatrix = array[0..TNColumnSize] of TNvector;
```

TNRowSize is an upper bound for the number of values returned for a particular variable (*NumReturn*). *TNColumnSize* is an upper bound for the number of second-order differential equations (*NumEquations*).

User-Defined Functions

```
function TNTargetF1(V : TNvector) : Real;
function TNTargetF2(V : TNvector) : Real;
function TNTargetF3(V : TNvector) : Real;
function TNTargetF4(V : TNvector) : Real;
function TNTargetF5(V : TNvector) : Real;
function TNTargetF6(V : TNvector) : Real;
function TNTargetF7(V : TNvector) : Real;
function TNTargetF8(V : TNvector) : Real;
function TNTargetF9(V : TNvector) : Real;
function TNTargetF10(V : TNvector) : Real;
```

Here are the differential equations:

$$d^2x_j/dt^2 = TNTargetFj(t, x_1, x'_1, x_2, x'_2, \dots, x_{10}, x'_{10})$$

where j ranges from 1 to 10.

The elements of the vector V are defined as follows:

$$\begin{aligned} V[0].x &= t \\ V[1].x &= x[1] \\ V[1].xDeriv &= x'[1] \\ V[2].x &= x[2] \\ V[2].xDeriv &= x'[2] \\ &\vdots \\ V[10].x &= x[10] \\ V[10].xDeriv &= x'[10] \end{aligned}$$

The procedure defined in RUNGE_S2.INC solves this system of coupled differential equations (a maximum of ten equations). All ten functions must be defined, even if your system contains less than ten equations.

Input Parameters

NumEquations : Integer;	Number of second-order differential equations
LowerLimit : Real;	Lower limit of interval
UpperLimit : Real;	Upper limit of interval
InitialValues : TNvector;	Values of x_j 's and x_j' 's at <i>LowerLimit</i>
NumReturn : Integer;	Number of $(t, x_1, x'_1, x_2, x'_2, \dots, x_m, x'_m)$ values returned from the procedure
NumIntervals : Integer;	Number of subintervals used in the calculations

The preceding parameters must satisfy the following conditions:

1. $NumReturn > 0$
2. $NumIntervals \geq NumReturn$
3. $NumEquations > 0$
4. $LowerLimit \neq UpperLimit$

Output Parameters

SolutionValues : TNmatrix; Values of t , x_j , and x'_j between the limits

Error : Byte;

- 0: No errors
- 1: $NumReturn < 1$
- 2: $NumIntervals < NumReturn$
- 3: $NumEquations < 1$
- 4: $LowerLimit = UpperLimit$

Syntax of the Procedure Call

```
InitialConditionSystem2(NumEquations, LowerLimit, UpperLimit,  
                        InitialValues, NumReturn, NumIntervals,  
                        SolutionValues, Error);
```

Comments

The first row of *SolutionValues* will be the values of t between the limits, the second row of *SolutionValues* will be the values of x_1 and x'_1 between the limits, the third row of *SolutionValues* will be the values of x_2 and x'_2 between the limits, and so on.

All ten user-defined functions are called from the procedure. If your system has less than ten equations, you must still define all ten functions or the program will not compile. The superfluous functions should be defined as follows (*TNTargetF10* is used as an example):

```
function TNTargetF10(V : TNvector) : Real;  
  
begin  
end;           { function TNTargetF10 }
```

If you need to solve a system with more than ten equations, then edit the include file `RUNGE_S2.INC`. The following lines should be added to the end of procedure *Step*:

```
F[11].xDeriv := Spacing * CurrentValues[11].xDeriv;  
F[11].x := Spacing * TNTargetF11(CurrentValues);
```

More statements (for $F[12]$, and so on) may be added as necessary. All new functions (for example, *TNTargetF11*) must be defined in your top-level program. **Note:** Before making any changes to the include file, make sure you have a backup copy.

The procedure will compute *NumIntervals* values in its calculations; however, you will rarely need to use these values. The rows of *SolutionValues* will contain only *NumReturn* values at roughly equally spaced t -values between the lower and upper limits. (They will be equally spaced only when *NumIntervals* is a multiple of *NumReturn*.) Thus, you can ensure a highly accurate solution (by making *NumIntervals* large) without generating an excessive amount of output (by making *NumReturn* small).

This routine stores much information on the heap. If you try to accurately solve a large system (that is, both *NumEquations* and *NumIntervals* are large), you may get run-time error \$FF, Heap/Stack collision. If this happens, the dimension of *TNvector* and *TNmatrix* should be reduced as much as possible. If this is not possible, then remove any RAM-resident software (for example, SideKick, SuperKey, or a print buffer).

The Runge-Kutta method uses the *New/Dispose* procedures to manipulate the heap and should not be used in any program that uses *Mark/Release* to manipulate the heap.

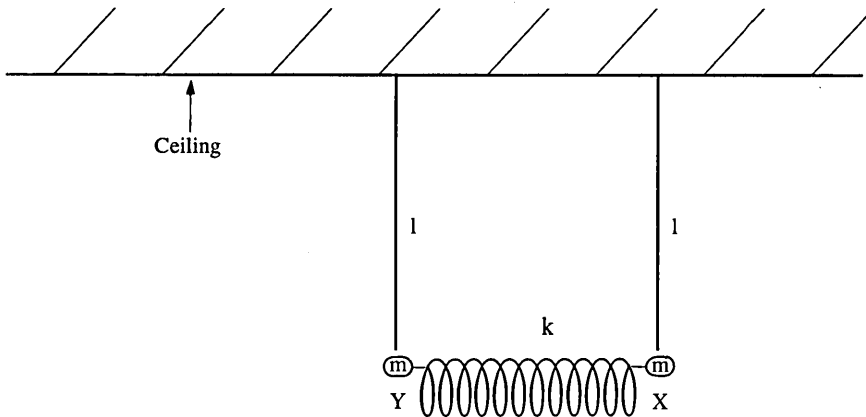
Warning: A stiff differential equation occurs when there are at least two very different scales of the independent variable on which the dependent variable(s) is changing; for example, $y = x + e^{-100x}$. The Runge-Kutta method may generate a numerical solution that bears no resemblance to the exact solution of the differential equation. This unstable numerical solution usually grows exponentially and may be oscillatory. However, if the exact solution of the differential equation grows as the independent variable increases, the instability may be difficult to detect. If a suspected instability has been encountered, reduce the interval size (*NumIntervals*).

Sample Program

The sample program RUNGE_S2.PAS provides I/O functions that demonstrate the Runge-Kutta method of solving initial value problems for systems of first-order ordinary differential equations. Note that the file RUNGE_S2.INC is included after the *TNTargetF* functions are defined.

Example

Problem. Two weights of mass m each hang from a pendulum of length l and are connected by a spring with spring constant k :



The equations of motion of these two masses are as follows:

$$m \frac{d^2 x}{dt^2} = -m g x/l - k(x - y)$$

$$m \frac{d^2 y}{dt^2} = -m g y/l + k(x - y)$$

where g is the acceleration due to gravity, t is time, and x and y are the displacements of the two weights from their rest positions. Given

$$m = 2 \text{ kg}$$

$$k = 32 \text{ N/m}$$

$$g = 9.8 \text{ m/sec}^2$$

$$l = 0.6125 \text{ m}$$

$$x(0) = 1 \text{ m}$$

$$y(0) = -1 \text{ m}$$

$$dx(0)/dt = 0 \text{ m/sec}$$

$$dy(0)/dt = 0 \text{ m/sec}$$

find the positions and velocities of the two weights from $t = 0$ second to $t = 2$ seconds.

1. Rewrite the equations of motion as shown here:

$$d^2x/dt^2 = -g x/l - k/m(x - y)$$

$$d^2y/dt^2 = -g y/l + k/m(x - y)$$

2. Code these equations into the program RUNGE_S2.PAS:

```
function TNTargetF1(V : TNvector) : Real;
(*****)
(*      THIS IS THE FIRST DIFFERENTIAL EQUATION      *)
(*****)
(*      *)
(*      *)
(*      d2 x[1] *)
(*      ----- = TNTargetF1(t, x[1], x'[1], x[2], x'[2], *)
(*      ..., x[m], x'[m] *)
(*      dt2 *)
(*      *)
(*      The elements of the vector V are defined: *)
(*      V[0].x = t *)
(*      V[1].x = X[1] *)
(*      V[1].xDeriv = X'[1] *)
(*      V[2].x = X[2] *)
(*      V[2].xDeriv = X'[2] *)
(*      . *)
(*      . *)
(*      . *)
(*      V[m].x = X[m] *)
(*      V[m].xDeriv = X'[m] *)
(*      *)
(*      where m is the number of coupled equations. *)
(*****)

var
  t : Real;

begin
  t := v[0].x;
  TNTargetF1 := -9.8 * V[1].x/0.6125 - 32/2 * (V[1].x - V[2].x);
end;
{ function TNTargetF1 }
```

```

function TNTargetF2(V : TNvector) : Real;

(*****)
(* THIS IS THE SECOND DIFFERENTIAL EQUATION *)
(*****)
(*
(*
(* d2x[2]
(* ----- = TNTargetF2(t, x[1], x'[1], x[2], x'[2],
(*          ..., x[m], x'[m]
(* dt2
(*
(*
(*
(* The elements of the vector V are defined:
(*   V[0].x = t
(*   V[1].x = X[1]
(*   V[1].xDeriv = X'[1]
(*   V[2].x = X[2]
(*   V[2].xDeriv = X'[2]
(*   .
(*   .
(*   .
(*   V[m].x = X[m]
(*   V[m].xDeriv = X'[m]
(*
(* where m is the number of coupled equations.
(*****)

var
  t : Real;

begin
  t := v[0].x;
  TNTargetF2 := -9.8 * V[2].x/0.6125 + 32/2 * (V[1].x - V[2].x);
end;
  { function TNTargetF2 }

```

```

function TNTargetF3(V : TNvector) : Real;

(*****
* THIS IS THE THIRD DIFFERENTIAL EQUATION *
*****)
(*
*)
(* d2 x[3]
* ----- = TNTargetF3(t, x[1], x'[1], x[2], x'[2],
*          ..., x[m], x'[m]
* dt2
*)
(*
*)
(* The elements of the vector V are defined:
*)
(*   V[0].x = t
*)
(*   V[1].x = X[1]
*)
(*   V[1].xDeriv = X'[1]
*)
(*   V[2].x = X[2]
*)
(*   V[2].xDeriv = X'[2]
*)
(*   .
*)
(*   .
*)
(*   .
*)
(*   V[m].x = X[m]
*)
(*   V[m].xDeriv = X'[m]
*)
(* where m is the number of coupled equations.
*)
(*****)

var
  t : Real;
begin
  { function TNTargetF3 }
end;

```

Functions *TNTargetF4* to *TNTargetF10* should be defined like function *TNTargetF3*.

3. Run RUNGE_S2.PAS:

Number of second order equations: (1-20)? 2

Lower limit of interval? 0

Upper limit of interval? 1

Enter X[1] value at t = 0.00000000E+00: 0.01

Enter X'[1] value at t = 0.00000000E+00: 0.00

Enter X[2] value at t = 0.00000000E+00: -0.01

Enter X'[2] value at t = 0.00000000E+00: 0.00

Number of values to return (1-100)? 10

Number of intervals (>= 10, default=10)? 100

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

Lower limit: 0.000000000000E+000
Upper limit: 1.000000000000E+000
Number of intervals: 100
Initial conditions at lower limit:
X[1]= 1.000000000000E-002
X'[1]= 0.000000000000E+000
X[2]= -1.000000000000E-002
X'[2]= 0.000000000000E+000

t	Value X[1]	Deriv X[1]
0.00000000	1.000000000000E-002	0.000000000000E+000
0.10000000	7.69447788485895E-003	-4.42511063153028E-002
0.20000000	1.84099813762452E-003	-6.80978317847279E-002
0.30000000	-4.86137387553900E-003	-6.05443464988731E-002
0.40000000	-9.32214486443693E-003	-2.50735962983904E-002
0.50000000	-9.48443369885917E-003	2.19586991271007E-002
0.60000000	-5.27340834792187E-003	5.88657408762406E-002
0.70000000	1.36920877108260E-003	6.86295294795966E-002
0.80000000	7.38047758874091E-003	4.674793932010E-002
0.90000000	9.98857556718864E-003	3.31066873866277E-003
1.00000000	7.99089728515028E-003	-4.16531651968366E-002

t	Value X[2]	Deriv X[2]
0.00000000	-1.000000000000E-002	0.000000000000E+000
0.10000000	-7.69447788485895E-003	4.42511063153028E-002
0.20000000	-1.84099813762452E-003	6.80978317847279E-002
0.30000000	4.86137387553900E-003	6.05443464988731E-002
0.40000000	9.32214486443693E-003	2.50735962983904E-002
0.50000000	9.48443369885917E-003	-2.19586991271007E-002
0.60000000	5.27340834792187E-003	-5.88657408762406E-002
0.70000000	-1.36920877108260E-003	-6.86295294795966E-002
0.80000000	-7.38047758874091E-003	-4.674793932010E-002
0.90000000	-9.98857556718864E-003	-3.31066873866277E-003
1.00000000	-7.99089728515028E-003	4.16531651968366E-002

The weights move in opposite directions; the system is in one of its normal modes. The natural frequency ω_o is given by the following:

$$\omega_o^2 = g/l + 2k/m$$

$$\omega_o = 6.928 \text{ cycles/sec}$$

Thus the period of oscillation, t , is

$$t = 2\pi/\omega_o$$

$$t = 0.9069 \text{ sec}$$

Solution to Boundary Value Problem for a Second-Order Ordinary Differential Equation Using the Shooting and Runge-Kutta Methods (SHOOT2.INC)

Description

This example uses the shooting method to approximate the solution to a second-order ordinary differential equation with specified boundary conditions (Burden and Faires 1985, 526–531).

Given a second-order differential equation (Burden and Faires 1985, 261–269) of the form

$$d^2y/dx^2 = \text{TNTargetF}(x, y, y')$$

where y' represents dy/dx , which satisfies the conditions given at the beginning of this chapter, boundary conditions

$$y[\text{LowerLimit}] = \text{LowerInitial}$$

$$y[\text{UpperLimit}] = \text{UpperInitial}$$

and spacing

$$h = (\text{UpperLimit} - \text{LowerLimit})/\text{NumIntervals}$$

and an initial approximation (guess) to the slope at *LowerLimit*

$$y'[\text{LowerLimit}] = \text{InitialSlope}$$

the shooting method first solves the second-order initial value problem (using the method described in RUNGE_2.INC). Based on a comparison of the solution at *UpperLimit* with the boundary condition *UpperInitial*, a new approximation to the slope at *LowerLimit* is made. In this way, a new “shot” at the solution is made by observing the result of the previous “shot.” Subsequent iterations use information from two previous shots and the secant method (see Chapter 2, “Roots of a Function Using the Secant Method”) to approximate the slope at *LowerLimit*. This process is repeated until the fractional difference between successive approximations to the boundary condition at *UpperLimit* is less than a specified tolerance.

You must supply the *LowerLimit*, *UpperLimit*, *LowerInitial*, *UpperInitial*, *InitialSlope*, *NumIntervals*, *Tolerance*, and *TNTargetF*.

User-Defined Types

TNvector = array[1..TNArraySize] of Real;

User-Defined Function

TNTargetF(x, y, yPrime : Real) : Real;

$$d^2y/dx^2 = \text{TNTargetF}(x, y, dy/dx)$$

The procedure *Shooting* integrates this second-order differential equation.

Input Parameters

LowerLimit : Real;	Lower limit of interval
UpperLimit : Real;	Upper limit of interval
LowerInitial : Real;	Value of y at <i>LowerLimit</i>
UpperInitial : Real;	Value of y at <i>UpperLimit</i>
InitialSlope : Real;	Approximation to the slope at <i>LowerLimit</i>
NumReturn : Integer;	Number of (x, y, y') values returned from the procedure
Tolerance : Real;	Indicates accuracy in solution
MaxIter : Integer;	Maximum number of iterations
NumIntervals : Integer;	Number of subintervals used in calculations

The preceding parameters must satisfy the following conditions:

1. $\text{NumReturn} > 0$
2. $\text{NumIntervals} \geq \text{NumReturn}$
3. $\text{LowerLimit} \neq \text{UpperLimit}$
4. $\text{Tolerance} > 0$
5. $\text{MaxIter} > 0$

Output Parameters

Iter : Integer;	Number of iterations required to reach a solution
XValues : TNvector;	Values of x between the limits
YValues : TNvector;	Values of y determined at values in <i>XValues</i>
YDerivValues : TNvector;	Values of the first derivative of y determined at values in <i>XValues</i>
Error : Byte;	0: No errors 1: $NumReturn < 1$ 2: $NumIntervals < NumReturn$ 3: $LowerLimit = UpperLimit$ 4: $Tolerance \leq 0$ 5: $MaxIter \leq 0$ 6: $Iter > MaxIter$ 7: Convergence not possible

Syntax of the Procedure Call

```
Shooting(LowerLimit, UpperLimit, LowerInitial, UpperInitial, InitialSlope,  
         NumReturn, Tolerance, MaxIter, NumIntervals, Iter, XValues,  
         YValues, YDerivValues, Error);
```

Comments

The parameter *Tolerance* can be misleading. The shooting method converges to the initial slope, which satisfies the upper boundary condition. Convergence is achieved when the fractional difference between *UpperInitial* and the upper boundary approximation is less than *Tolerance*. This does not mean that every value between the boundaries has been approximated with the same degree of accuracy. To improve the accuracy of the entire approximation, increase the number of intervals. The example demonstrates the different effects of *Tolerance* and *NumIntervals*.

The shooting algorithm will compute *NumIntervals* values in its calculations. However, you will rarely need to use all those values. The vectors *XValues*, *YValues*, and *YDerivValues* will contain only *NumReturn* values at roughly equally spaced t -values between the lower and upper limits. (They will be equally spaced only when *NumIntervals* is a multiple of *NumReturn*.) Thus, you can ensure a highly accurate solution (by making *NumIntervals* large) without generating an excessive amount of output (by making *NumReturn* small).

Boundary value problems are notoriously difficult to solve. The shooting method is extremely sensitive to the approximation of the initial slope. If the shooting method does not converge onto a solution (Error 7), run the program with a different value of the initial slope *InitialSlope*. You may also alleviate some stability problems by solving the equation backwards (from *UpperLimit* to *LowerLimit*). Considerable trial and error may be involved before a solution is found.

The Runge-Kutta method uses the *New/Dispose* procedures to manipulate the heap and should not be used in any program that uses *Mark/Release* for heap management.

Warning: A stiff differential equation occurs when there are at least two very different scales of the independent variable on which the dependent variable(s) is changing; for example, $y = x + e^{100x}$. The shooting method may generate a numerical solution that bears no resemblance to the exact solution of the differential equation. This unstable numerical solution usually grows exponentially and may be oscillatory. However, if the exact solution of the differential equation grows as the independent variable increases, the instability may be difficult to detect. If a suspected instability has been encountered, reduce the interval size (*NumIntervals*).

Sample Program

The sample program SHOOT2.PAS provides I/O functions that demonstrate the shooting method of solving boundary value problems. Note that the file SHOOT2.INC is included after the function *TNTargetF* is defined.

Example

Problem. Use the nonlinear shooting method to solve the following boundary value problem:

$$y'' = 192 \operatorname{sqr}(y/y') \quad 0 \leq x \leq 1$$

$$y(1) = 1$$

$$y(2) = 16$$

1. Code the differential equation into the program:

```
function TNTargetF(x : Real;  
                  y : Real;  
                  yPrime : Real) : Real;  
  
  (*****  
  (*   THIS IS THE SECOND-ORDER NONLINEAR DIFFERENTIAL EQUATION   *)  
  (*****  
  
begin  
  TNTargetF := 192 * Sqr(y/yPrime);  
end;                                     {function TNTargetF}
```

2. Run SHOOT2.PAS:

Lower limit of interval? 0

Upper limit of interval? 1

Enter Y value at X = 0.0000000E+00: 1

Enter Y value at X = 1.0000000E+00: 16

Enter a guess for the slope at X = 0.0000000E+00 (default=1.50E+01): 15

Number of points returned (1-500)? 10

Number of intervals (>= 10, default=10)? 10

Tolerance (> 0, default = 1.000E-06)? 1E-12

Maximum number of iterations (> 0, default = 100)? 100

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

```
          Lower limit: 0.0000000000000E+000  
          Upper limit: 1.0000000000000E+000  
    Value of Y at 0.0000: 1.0000000000000E+000  
    Value of Y at 1.0000: 1.6000000000000E+001  
Initial slope at 0.0000: 1.5000000000000E+001  
      NumIntervals: 10  
        Tolerance: 1.000E-012  
Maximum number of iterations: 100
```

Number of iterations: 6

X	Y Value	Derivative of Y
0.000000000000000E+000	1.000000000000000E+000	4.00053795390884E+000
1.000000000000000E-001	1.46417721408153E+000	5.32386904044879E+000
2.000000000000000E-001	2.07370562259973E+000	6.91162114244397E+000
3.000000000000000E-001	2.85621262766442E+000	8.78752756627335E+000
4.000000000000000E-001	3.84170902091389E+000	1.09754927855527E+001
5.000000000000000E-001	5.06259931530967E+000	1.34994802016423E+001
6.000000000000000E-001	6.55368547624580E+000	1.63834750611955E+001
7.000000000000000E-001	8.35216836918581E+000	1.96514712240017E+001
8.000000000000000E-001	1.04976483580762E+001	2.33274661179548E+001
9.000000000000000E-001	1.30321255669365E+001	2.74354587043771E+001
1.000000000000000E+000	1.60000000000094E+001	3.19994486182108E+001

Now solve the same problem using a smaller spacing, but with a greater tolerance:

Lower limit of interval? 0

Upper limit of interval? 1

Enter Y value at X = 0.00000000E+00: 1

Enter Y value at X = 1.00000000E+00: 16

Enter a guess for the slope at X = 0.00000000E+00 (default = 1.50E+01): 15

Number of points returned (1-500)? 10

Number of intervals (≥ 10 , default = 10)? 100

Tolerance (> 0 , default = 1.000E-06)? 1E-6

Maximum number of iterations (> 0 , default = 100)? 100

Direct output to one of the following:

(S)creen

(P)rinter

(F)ile

```
Lower limit: 0.000000000000000E+000
Upper limit: 1.000000000000000E+000
Value of Y at 0.0000: 1.000000000000000E+000
Value of Y at 1.0000: 1.600000000000000E+001
Initial slope at 0.0000: 1.500000000000000E+001
NumIntervals: 100
Tolerance: 1.000E-006
Maximum number of iterations: 100
```

Number of iterations: 5

X	Y Value	Derivative of Y
0.000000000000000E+000	1.000000000000000E+000	4.00000062625638E+000
1.000000000000000E-001	1.46410005120828E+000	5.32400035609946E+000
2.000000000000000E-001	2.07360008576235E+000	6.91200027103432E+000
3.000000000000000E-001	2.85610011557157E+000	8.78800025750412E+000
4.000000000000000E-001	3.84160014547825E+000	1.09760002747783E+001
5.000000000000000E-001	5.06250017769403E+000	1.35000003070170E+001
6.000000000000000E-001	6.55360021337284E+000	1.63840003476283E+001
7.000000000000000E-001	8.35210025321451E+000	1.96520003937080E+001
8.000000000000000E-001	1.04976002977125E+001	2.33280004439140E+001
9.000000000000001E-001	1.30321003472617E+001	2.74360004976014E+001
1.000000000000000E+000	1.60000004022081E+001	3.20000005544507E+001

The exact solution is

$$y = (x + 1)^4$$

X	Y Value	Derivative of Y
0.0	1.0000000000	4.0000000000
0.1	1.4641000000	5.3240000000
0.2	2.0736000000	6.9120000000
0.3	2.8561000000	8.7880000000
0.4	3.8416000000	1.0976000000
0.5	5.0625000000	1.3500000000
0.6	6.5536000000	1.6384000000
0.7	8.3521000000	1.9652000000
0.8	1.0497600000	2.3328000000
0.9	1.3032100000	2.7436000000
1.0	1.6000000000	3.2000000000

Although the tolerance is smaller (that is, more exacting) in the first case, the accuracy of the approximation is greater in the second case. The spacing in the first case is too large to permit a more accurate approximation.

Solution to a Boundary Value Problem for a Second-Order Ordinary Linear Differential Equation Using the Linear Shooting and Runge-Kutta Methods (LINSHOT2.INC)

Description

This example uses the linear shooting method to approximate the solution to a second-order linear ordinary differential equation with specified boundary conditions (Burden and Faires 1985, 519–524).

Given a second-order differential equation (Burden and Faires 1985, 261–264) of the form

$$d^2y/dx^2 = TNTargetF(x, y, y')$$

which is linear in both y and y' , where y' represents dy/dx , and which satisfies the conditions given at the beginning of this chapter, boundary conditions

$$y[LowerLimit] = LowerInitial$$

$$y[UpperLimit] = UpperInitial$$

and spacing

$$h = (UpperLimit - LowerLimit)/NumIntervals$$

the shooting method solves the two initial value problems (see RUNGE_2.INC):

$$y'[LowerLimit] = 0 \qquad y[LowerLimit] = LowerInitial$$

$$y'[LowerLimit] = 1 \qquad y[LowerLimit] = LowerInitial$$

(These values are particular to this implementation; any other nonidentical set of initial conditions will suffice.) Since neither of these initial values of y' is likely to be correct, the solutions generated are not likely to satisfy the boundary condition at $UpperInitial$. However, because of the linearity of the equation, an appropriate linear combination of these two solutions will be a solution to the boundary value problem. The linear shooting method requires that only two initial value problems be solved, where the ordinary shooting method (SHOOT2.INC) requires that an unknown number of initial value problems be solved before the method converges to a solution.

You must supply the *LowerLimit*, *UpperLimit*, *LowerInitial*, *UpperInitial*, *NumIntervals*, and *TNTargetF*.

User-Defined Types

TNvector = array[1..TNArraySize] of Real;

User-Defined Function

TNTargetF(x, y, yPrime : Real) : Real;

$$d^2y/dx^2 = \text{TNTargetF}(x, y, dy/dx)$$

The procedure *LinearShooting* integrates this second-order differential equation.

Input Parameters

LowerLimit : Real;	Lower limit of interval
UpperLimit : Real;	Upper limit of interval
LowerInitial : Real;	Value of y at <i>LowerLimit</i>
UpperInitial : Real;	Value of y at <i>UpperLimit</i>
NumIntervals : Integer;	Number of subintervals used in calculations
NumReturn : Integer;	Number of (x, y, y') triples returned from the procedure

The preceding parameters must satisfy the following conditions:

1. $\text{NumReturn} > 0$
2. $\text{NumIntervals} \geq \text{NumReturn}$
3. $\text{LowerLimit} \neq \text{UpperLimit}$

Output Parameters

XValues : TNvector;	Values of x between the limits
YValues : TNvector;	Values of y determined at values in <i>XValues</i>
YDerivValues : TNvector;	Values of the first derivative of y determined at values in <i>XValues</i>
Error : Byte;	0: No errors 1: $\text{NumReturn} < 1$ 2: $\text{NumIntervals} < \text{NumReturn}$ 3: $\text{LowerLimit} = \text{UpperLimit}$ 4: Equation not linear

Syntax of the Procedure Call

```
LinearShooting(LowerLimit, UpperLimit, LowerInitial, UpperInitial,  
              NumReturn, NumIntervals, XValues, YValues,  
              YDerivValues, Error);
```

Comments

If *TNTargetF* is a nonlinear function, the linear shooting algorithm will usually compute a solution (albeit an incorrect one) without returning an error message. Error 4 (nonlinear equation) will be returned in only a few cases where the two initial value problems happen to yield solutions with the same *y*-value at $x = \text{UpperLimit}$.

The procedure will compute *NumIntervals* values in its calculations; however, you will rarely need to use these values. The vectors *XValues*, *YValues*, and *YDerivValues* will contain only *NumReturn* values at roughly evenly spaced intervals between the lower and upper limits. (They will be exactly evenly spaced only when *NumIntervals* is a multiple of *NumReturn*.) Thus, you can ensure a highly accurate solution (by making *NumIntervals* large) without generating an excessive amount of output (by making *NumReturn* small).

The Linear Shooting/Runge Kutta method uses the *New/Dispose* procedures to manipulate the heap and should not be used in any program that uses *Mark/Release* to manipulate the heap.

Warning: A stiff differential equation occurs when there are at least two very different scales of the independent variable on which the dependent variable(s) is changing; for example, $y = x + e^{-100x}$. The Runge-Kutta method may generate a numerical solution that bears no resemblance to the exact solution of the differential equation. This unstable numerical solution usually grows exponentially and may be oscillatory. However, if the exact solution of the differential equation grows as the independent variable increases, the instability may be difficult to detect. If a suspected instability has been encountered, reduce the interval size (*NumIntervals*).

Sample Program

The sample program LINSHOT2.PAS provides I/O functions that demonstrate the linear shooting method of solving boundary value problems. Note that the file LINSHOT2.INC is included after the function *TNTargetF* is defined.

Example

Problem. Use the linear shooting method to solve the following boundary value problem:

$$\begin{aligned}y'' &= y'/x - y/\text{sqr}(x) + 1 & 1 &= x \leq 10 \\ y(1) &= 1 \\ y(10) &= 76.974149\end{aligned}$$

1. Code the differential equation into the program LINSHOT2.PAS:

```
function TNTargetF(x : Real;
                  y : Real;
                  yPrime : Real) : Real;

(*****
(*)      THIS IS THE SECOND-ORDER DIFFERENTIAL EQUATION      *)
(*****)

begin
  TNTargetF := yPrime/x - y/Sqr(x) + 1;
end;      { function TNTargetF }
```

2. Run LINSHOT2.PAS:

```
Lower limit of interval? 1
Upper limit of interval? 10

Enter Y value at X = 1.00000000E+00: 1
Enter Y value at X = 1.00000000E+01: 76.974149

Number of points returned (1-500)? 9

Number of intervals (>= 9, default = 9)? 9

Direct output to one of the following:
  (S)creen
  (P)rinter
  (F)ile

      Lower limit: 1.00000000000000E+00
      Upper limit: 1.00000000000000E+01
Value of Y at 1.0000: 1.00000000000000E+00
Value of Y at 10.0000: 7.69741490000000E+01
      NumIntervals: 9.00000000000000E+00
```

X	Y Value	Derivative of Y
1.00000000000000E+000	1.00000000000000E+000	1.00042467674563E+000
2.00000000000000E+000	2.61170356138588E+000	2.30627678512124E+000
3.00000000000000E+000	5.70207271413620E+000	3.90115296191831E+000
4.00000000000000E+000	1.04528257144925E+001	5.61367861126495E+000
5.00000000000000E+000	1.69509897305375E+001	7.39067355864438E+000
6.00000000000000E+000	2.52478687612139E+001	9.20845513089500E+000
7.00000000000000E+000	3.53773649984557E+001	1.10543869346579E+001
8.00000000000000E+000	4.73635728977226E+001	1.29209245920937E+001
9.00000000000000E+000	6.12245068576119E+001	1.48032011472994E+001
1.00000000000000E+001	7.69741490000000E+001	1.66978931711222E+001

Now solve the same problem with a spacing of only 0.1:

Lower limit of interval? 1

Upper limit of interval? 10

Enter Y value at X = 1.00000000E+00: 1

Enter Y value at X = 1.00000000E+01: 76.974149

Number of points returned (1-500)? 9

Number of intervals (>= 9, default = 9)? 90

Direct output to one of the following:

(S)creen

(P)rinter

(F)ile

Lower limit: 1.00000000000000E+00
Upper limit: 1.00000000000000E+01
Value of Y at 1.0000: 1.00000000000000E+00
Value of Y at 10.0000: 7.69741490000000E+01
NumIntervals: 9.00000000000000E+01

X	Y Value	Derivative of Y
1.00000000000000E+000	1.00000000000000E+000	1.00000001942594E+000
2.00000000000000E+000	2.61370547174514E+000	2.30685275847028E+000
3.00000000000000E+000	5.70416298088411E+000	3.90138768358927E+000
4.00000000000000E+000	1.04548224122436E+001	5.61370562650429E+000
5.00000000000000E+000	1.69528103026793E+001	7.39056208402440E+000
6.00000000000000E+000	2.52494430584438E+001	9.20824053324639E+000
6.99999999999999E+000	3.53786288412165E+001	1.10540898579729E+001
7.99999999999999E+000	4.73644675641047E+001	1.29205584690303E+001
8.99999999999999E+000	6.12249787166508E+001	1.48027754364805E+001
9.99999999999998E+000	7.69741490000000E+001	1.66974149235206E+001

The exact solution is

$$y = x * x - x * \ln(x)$$

$$\begin{aligned} y(1) &= 1 & y'(1) &= 1 \\ y(10) &= 7.6974149 & y'(10) &= 16.6974149 \end{aligned}$$

The second approximation is more accurate.

Least-Squares Approximation

Given a set of data points, this chapter provides routines to model the data with a function of a given type. The most common application of this concept is *linear regression*.

With linear regression, there is some control variable, say X , and some observed variable, say Y . X and Y are known or suspected to have some linear relationship, say

$$Y = a * X + b$$

but the parameters a and b are unknown. Usually there is some experimental error or some other nonlinear influence on Y , so that there are no values of a and b for which the preceding equation holds exactly. The method of regression provides a formula for a and b in terms of the values of X and Y such that the error is minimized. The error is the sum of squares of the errors ($a * X + b - Y$) on each data point. Except in certain unusual cases, there is exactly one value for a and one value for b that makes this sum of squares the least possible. This is called the *least-squares* solution.

This concept of least squares also applies when more variables are present—then it is often called *multiple regression*. Using this method, you can find the best model for a given set of data that is linear in a given set of other data sets or functions. Models that are nonlinear variables can also be treated as long as the unknown parameters appear linearly.

Least-Squares Approximation (LEAST.INC)

Description

This model provides a method for finding a least-squares approximation (Cheney and Kincaid 1985, 362–387) to a set of data points (x, y) . The approximation must be a linear combination of a set of *basis vectors*. The functional form of the approximation (polynomial, logarithmic, and so on) is therefore determined by the user, as long as it is represented linearly. (How to represent logarithmic, and other functions linearly is discussed later.)

Given a set of m data points (x, y) , an $m \times n$ matrix ($m \geq n$), A , is constructed, where n is the number of basis vectors in the approximation. The elements of the matrix are

$$A[i, j] = V_j(X_i)$$

where $V_j(X_i)$ is the j th basis vector evaluated at the data value $X[i]$. A vector Y is constructed that contains the y -values of the data points. The coefficients of the basis vectors that form the least-squares approximation will be the n vector C , such that the Euclidean norm of $(AC - Y)$ (represented by $\|AC - Y\|_2$) is a minimum. This requirement is converted to the requirement that

$$\|BC - Z\|_2 + \|R\|_2$$

be a minimum. Here B is an $n \times n$ matrix, Z is an n vector, and R is an $(m - n)$ vector. The equations $BC = Z$ are the normal equations. The previous expression will be minimized when C solves the equation $BC = Z$. Gaussian elimination with partial pivoting (see Chapter 6, “Solving a System of Linear Equations with Gaussian Elimination and Partial Pivoting”) is used to solve the normal equations.

The *goodness of fit* is indicated by the standard deviation:

$$\text{S.D.} = ((Y[i] - F(X[i]))^2 / (m - n))^{1/2}$$

where $F(X[i])$ is the least-squares solution at the point $X[i]$, $(Y[i] - F(X[i]))$ is the residual, and $(m - n)$ is the degree of freedom of the fit.

User-Defined Types

```
TNColumnVector = array[1..TNColumnSize] of Real;  
TNRowVector = array[1..TNRowSize] of Real;
```

(*TNColumnSize* will usually be much larger than *TNRowSize*.)

```
TNmatrix = array[1..TNColumnSize] of TNRowVector;  
TNSquareMatrix = array[1..TNRowSize] of TNRowVector;  
TNString40 = string[40];
```

Input Parameters

NumPoints : Integer; Number of data points
XData : TNColumnVector; *X* coordinates of the data points
YData : TNRowVector; *Y* coordinates of the data points
NumTerms : Integer; Number of terms in the least-squares approximation

The preceding parameters must satisfy the following conditions:

1. *NumPoints* > 1.
2. *NumTerms* ≤ *NumPoints*.
3. *NumPoints* ≤ *TNColumnSize*.
4. *NumTerms* ≤ *TNRowSize*.
5. The *XData* points cannot all be identical.

TNColumnSize and *TNRowSize* set an upper bound on the number of elements in a vector. Neither of these identifiers are variable names and are never referenced by the procedure. If conditions 3 or 4 are violated, the program will crash with an Index Out of Range error (assuming the directive `{R+}` is active).

Output Parameters

Solution : TNRowVector;	Coefficients of the basis vectors that form the least-squares approximation
YFit : TNColumnVector;	Values of the least-squares fit evaluated at the <i>XData</i> values
Residuals : TNColumnVector;	Difference between <i>YData</i> and <i>YFit</i> values
StandardDeviation : Real;	Square root of the variance—indicates the goodness of fit
Error : Byte;	0: No error 1: <i>NumPoints</i> < 2 2: <i>NumTerms</i> < 1 3: <i>NumTerms</i> > <i>NumPoints</i> 4: Least-squares solution does not exist (see “Comments”)

Syntax of the Procedure Call

```
LeastSquares(NumPoints, XData, YData, NumTerms, Solution,  
             YFit, Residuals, StandardDeviation, Error);
```

Comments

The least-squares routine is kept in two modules (include files). One is called LEAST.INC and must always be included in your top-level program. The choice of the second module will depend upon the functional form (basis vectors) to which you fit the data. Following are the five basis modules included in this package:

POLY.LSQ

This module uses *Chebyshev polynomials* to fit a polynomial to the data points. *NumTerms* must be one greater than the degree of the polynomial (for example, to fit a fourth-degree polynomial, input *NumTerms* = 5). To get a straight-line least-squares fit, use this module and fit a curve with only two coefficients. The elements of the *Solution* vector will be as follows:

$$\text{Solution}[j] = a_j \quad 1 \leq j \leq \text{NumTerms}$$

where a_j is the coefficient of x^{j-1} .

FOURIER.LSQ

This module will fit a finite Fourier series to the data points. The number of terms in the approximation will be *NumTerms*. The elements of the *Solution* vector will be as follows:

$$\text{Solution}[j] = F_{j-1} \quad 1 \leq j \leq \text{NumTerms}$$

where F_{j-1} is the $(j-1)$ th term in the Fourier series. Following are the first few terms in the Fourier series:

$$F[0] = 1$$

$$F[1] = \cos(x)$$

$$F[2] = \sin(x)$$

$$F[3] = \cos(2x)$$

$$F[4] = \sin(2x)$$

$$F[5] = \cos(3x)$$

$$F[6] = \sin(3x)$$

POWER.LSQ

This module will fit the function

$$y = ax^b$$

where a and b are real numbers to the data points. A linear equation is obtained by taking the log of both sides, like so:

$$\ln(y) = \ln(a) + b * \ln(x)$$

and expanding on basis vectors 1 and $\ln(x)$. The x -values of the data points must all be positive, and the y -values of the data points must all have the same sign. The number of coefficients in the approximation will be two regardless of the value of *NumTerms* (unless *NumTerms* > *NumPoints*, in which case Error 3 will occur). The elements of the *Solution* vector will be as follows:

$$\text{Solution}[1] = a$$

$$\text{Solution}[2] = b$$

EXP.LSQ

This module will fit the function

$$y = ae^{bx}$$

where a and b are real numbers to the data points. A linear equation is obtained by taking the log of both sides, like so:

$$\ln(y) = \ln(a) + bx$$

and expanding on basis vectors 1 and x . The y -values of the data points must all have the same sign. The number of coefficients in the approximation will be two regardless of the value of *NumTerms* (unless *NumTerms* > *NumPoints*, in which case Error 3 will occur). The elements of the *Solution* vector will be as follows:

$$\begin{aligned} \text{Solution}[1] &= a \\ \text{Solution}[2] &= b \end{aligned}$$

LOG.LSQ

This module will fit the function

$$y = a \ln(bx)$$

where a and b are real numbers to the data points. A linear equation is obtained by rewriting the equation:

$$y = a \ln(b) + a \ln(x)$$

and expanding on basis vectors 1 and $\ln(x)$. The x -values of the data points must all have the same sign. The number of coefficients in the approximation will be two regardless of the value of *NumTerms* (unless *NumTerms* > *NumPoints*, in which case Error 3 will occur). The elements of the *Solution* vector will be as follows:

$$\begin{aligned} \text{Solution}[1] &= a \\ \text{Solution}[2] &= b \end{aligned}$$

USER.LSQ

This module is included if you need a least-squares approximation on a set of basis vectors different from the ones listed earlier. This module allows you to create your own set of basis vectors. The source code contains detailed instructions of how to flesh out the skeleton contained in USER.LSQ.

A least-squares solution may not exist for some input data and choice of basis vectors (Error 4). The reasons for this will depend on the module you are using. For example, it is impossible to fit an exponential function (module EXP.LSQ) to data with y -values of differing signs; Error 4 will occur if you try. The same data could be fit with a polynomial and no error would result. Error 4 will also occur if all the x -values of the data are identical.

Sample Program

The demonstration program LEAST.PAS contains I/O routines that allow you to run the least-squares approximation routine. Note that there are two include commands:

```
{ $I POLY.LSQ } (* load the basis vectors *)  
{ $I LEAST.INC } (* load procedure LeastSquares *)
```

The LEAST.INC file must always be included after the basis module. To change the basis vectors of the approximation, simply load a different basis module with the first INCLUDE command.

Input Files

Data may be entered from a text file. The x - and y -coordinates should be separated by a space and followed by a carriage return. For example, data values of $\text{sqr}(x)$ could be entered in a text file as

```
1 1  
2 4  
3 9  
4 16  
5 25
```

Example

Problem. Given the following data (contained in the file SAMPLE9A.DAT), fit a fourth-degree polynomial and a logarithmic function to the data:

0.00000000000000E+00	1.33830225764886E-03
0.10000000000000E+00	4.43184841193803E-02
0.20000000000000E+00	5.39909665131879E-01
0.30000000000000E+00	2.41970724519143E+00
0.40000000000000E+00	3.98942280401433E+00
0.02000000000000E+00	2.91946925791461E-03
0.04000000000000E+00	6.11901930113775E-03
0.06000000000000E+00	1.23221916847303E-02
0.08000000000000E+00	2.38408820146486E-02
0.12000000000000E+00	7.91545158298001E-02
0.14000000000000E+00	1.35829692336855E-01
0.16000000000000E+00	2.23945302948430E-01
0.18000000000000E+00	3.54745928462313E-01
0.22000000000000E+00	7.89501583008939E-01
0.24000000000000E+00	1.10920834679455E+00
0.26000000000000E+00	1.49727465635745E+00
0.28000000000000E+00	1.94186054983213E+00
0.32000000000000E+00	2.89691552761483E+00
0.34000000000000E+00	3.33224602891800E+00
0.36000000000000E+00	3.68270140303323E+00
0.38000000000000E+00	3.91042693975456E+00

(The function is the left-hand side of a Gaussian distribution curve with mean = 0.5 and standard deviation = 0.1.) Note that the points do not have to be in any particular order.

First fit the polynomial; include the proper include files in the LEAST.PAS program.

```
{ $I POLY.LSQ }      (* load the basis vectors *)
{ $I LEAST.INC }     (* load procedure LeastSquares *)
```

Run LEAST.PAS:

(K)eyboard or (F)ile entry of data? F

File name? SAMPLE9A.DAT

Number of terms in the least squares fit (<= 21)? 5

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

The Data Points:

X	Y
0.200	0.0443185
0.300	0.5399097
0.400	2.4197072
0.500	3.9894228
0.100	0.0013383
0.120	0.0029195
0.140	0.0061190
0.160	0.0123222
0.180	0.0238409
0.220	0.0791545
0.240	0.1358297
0.260	0.2239453
0.280	0.3547459
0.320	0.7895016
0.340	1.1092083
0.360	1.4972747
0.380	1.9418605
0.420	2.8969155
0.440	3.3322460
0.460	3.6827014
0.480	3.9104269

```
*-----*
Polynomial Least Squares Fit
*-----*
```

Coefficients in least squares approximation:

Coefficient 0: -3.1905595419E+00
 Coefficient 1: 6.4048009604E+01
 Coefficient 2: -4.3900537685E+02
 Coefficient 3: 1.2058567475E+03
 Coefficient 4: -1.0523352671E+03

X	Least Squares Fit	Residual
0.2000	2.1944857683E-02	-2.2373626436E-02
0.3000	5.4757594258E-01	7.6662774545E-03
0.4000	2.4228330082E+00	3.1257630399E-03
0.5000	4.0432402964E+00	5.3817492408E-02
0.1000	-7.5189129229E-02	-7.6527431486E-02
0.1200	3.9032402623E-02	3.6112933365E-02
0.1400	7.6262215339E-02	7.0143196038E-02
0.1600	6.8115144530E-02	5.5792952845E-02
0.1800	4.2165058391E-02	1.8324176377E-02
0.2200	2.6946475745E-02	-5.2208040084E-02
0.2400	7.2620878494E-02	-6.3208813842E-02
0.2600	1.7037806441E-01	-5.3567238538E-02
0.2800	3.2758706456E-01	-2.7158863898E-02
0.3200	8.2963179468E-01	4.0130211675E-02
0.3400	1.1690007497E+00	5.9792402863E-02
0.3600	1.5568879689E+00	5.9613312500E-02
0.3800	1.9804576462E+00	3.8597096373E-02
0.4200	2.8630963140E+00	-3.3819213604E-02
0.4400	3.2762888552E+00	-5.5957173721E-02
0.4600	3.6334109560E+00	-4.9290447012E-02
0.4800	3.9014219733E+00	-9.0049664504E-03

Standard Deviation : 5.381534E-02

The fourth-degree polynomial that best fits this data is as follows:

$$y = -1052.34 x^4 + 1205.86 x^3 - 439.005 x^2 + 64.0480 x - 3.19056$$

Note that a fourth-degree polynomial requires five terms in the fit.

Now fit the logarithmic function; include the proper include files in the LEAST.PAS program.

```
{I LOG.LSQ}      (* load the basis vectors *)
{I LEAST.INC}    (* load procedure LeastSquares *)
```

Run LEAST.PAS:

(K)eyboard of (F)ile entry of data? F

File name? SAMPLE9A.DAT

Number of terms in the least squares fit (<= 21)? 2

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

The Data Points:

X	Y
0.200	0.0443185
0.300	0.5399097
0.400	2.4197072
0.500	3.9894228
0.100	0.0013383
0.120	0.0029195
0.140	0.0061190
0.160	0.0123222
0.180	0.0238409
0.220	0.0791545
0.240	0.1358297
0.260	0.2239453
0.280	0.3547459
0.320	0.7895016
0.340	1.1092083
0.360	1.4972747
0.380	1.9418605
0.420	2.8969155
0.440	3.3322460
0.460	3.6827014
0.480	3.9104269

Logarithmic Least Squares Fit

Coefficients in least squares approximation:

Coefficient 0: 2.5984092387E+00

Coefficient 1: 6.0253489685E+00

X	Least Squares Fit	Residual
0.2000	4.8470072529E-01	4.4038224117E-01
0.3000	1.5382650082E+00	9.9835534304E-01
0.4000	2.2857807630E+00	-1.3392648216E-01
0.5000	2.8655990283E+00	-1.1238237757E+00
0.1000	-1.3163793125E+00	-1.3177176147E+00
0.1200	-8.4263329487E-01	-8.4555276412E-01
0.1400	-4.4208674425E-01	-4.4820576355E-01
0.1600	-9.5117540004E-02	-1.0743973169E-01
0.1800	2.1093098801E-01	1.8709010600E-01
0.2200	7.323557703E-01	6.5320106120E-01
0.2400	9.5844674288E-01	8.2261705054E-01
0.2600	1.1664304540E+00	9.4248515104E-01
0.2800	1.3589932935E+00	1.0042473650E+00
0.3200	1.7059624977E+00	9.1646091473E-01
0.3400	1.8634900752E+00	7.5428172837E-01
0.3600	2.0120110257E+00	5.1473636940E-01
0.3800	2.1524997930E+00	2.1063924317E-01
0.4200	2.4125575764E+00	-4.8435795124E-01
0.4400	2.5334356148E+00	-7.9881041414E-01
0.4600	2.6489394853E+00	-1.0337619177E+00
0.4800	2.7595267806E+00	-1.1509001591E+00

Standard Deviation : 8.320742E-01

The logarithmic function that bests fits this data is as follows:

$$y = 2.59841 * \ln(6.02535x)$$

The standard deviation of the polynomial fit is much smaller than that of the logarithmic fit; a fourth-degree polynomial fits this data much better than a logarithmic function.

Fast Fourier Transform Routines

Fourier transforms are used to analyze periodic phenomena such as waves. A continuous function f that has period 2π ($= 2 * 3.14159265\dots$); that is, satisfies

$$f(x + 2\pi) = f(x)$$

for all x , can be decomposed into sines and cosines:

$$f(x) = a[0] + a[1] * \cos(x) + b[1] * \sin(x) + a[2] * \cos(2x) \\ + b[2] * \sin(2x) + \dots$$

This is an infinite series where the coefficients get closer and closer to zero. The routines in this chapter can be used to calculate the coefficients.

The *Fast Fourier Transform* (FFT) is a particular algorithm for computing Fourier transforms efficiently.

This chapter includes two kinds of units. One group consists of four variations of the FFT method of calculating discrete Fourier transforms (FFTB2.INC, FFTB4.INC, FFT87B2.INC, FFT87B4.INC), each optimized for certain conditions. All are variations of the original *Cooley-Tukey* method. The second group consists of six applications (COMPFFT.INC, REALFFT.INC, COMPCNVL.INC, REALCNVL.INC, COMPCORR.INC, REALCORR.INC) that can each be used with any of the FFT methods. You can select the FFT method most appropriate to the circumstances and combine it with the appropriate application or integrate it into another program (Brigham 1974; Nussbaumer 1982).

In each FFT unit the procedure calls have exactly the same form (although there are different restrictions on the data) so that any one FFT unit can be combined

with any of the application units without rewriting code. Each of these algorithms will compute either a *forward* or an *inverse transform*.

Each unit contains two procedures needed to prepare for the FFT calculation: procedure *TestInput* and procedure *MakeSinCosTable*. *TestInput* examines the input data to ensure that it satisfies certain conditions (for example, that there is more than 1 data point). *MakeSinCosTable* precalculates a table of the n th roots of unity for look up in the FFT calculation.

FFTB2.INC contains a procedure that implements the Cooley-Tukey *powers-of-two* (*radix 2* or *base 2*) Fast Fourier Transforms. It is optimized to reduce the number of real multiplications by taking advantage of the symmetries of certain roots of unity and by using a complex multiplication that requires three real multiplications and three real additions. This algorithm is appropriate when the time for real multiplications is significantly greater than the time for real additions; for example, when running on an 8088 machine with no numeric coprocessor.

FFT87B2.INC implements the same algorithm as FFTB2.INC. The difference is that the complex multiplications are done with four real multiplications and two real additions. By using this standard form of complex multiplication, storage overhead and assignment statements are reduced. This algorithm is appropriate when the time for a real multiplication is close to the time for a real addition; for example, when running an 8088 machine with an 8087 numeric coprocessor, or an 80286 machine with an 80287 numeric coprocessor.

Standard Turbo Pascal uses 6-byte reals; Turbo-87 Pascal (which utilizes the 8087 coprocessor) uses 8-byte reals. Consequently, given the same amount of memory, more data points can be manipulated in Turbo Pascal than in Turbo-87 Pascal. Both FFTB2.INC and FFT87B2.INC require the number of data points to be a power of two up to a maximum of 4,096 points when in Turbo-87, or 8,192 points when in standard Turbo Pascal.

FFTB4.INC and FFT87B4.INC implement *powers-of-four* (*radix 4* or *base 4*) Fast Fourier Transforms. The powers-of-four method is the same as the Cooley-Tukey algorithm except at each stage of reduction a given transform is converted into four transforms each with one fourth the data points of its predecessor (Nus-sbaumer 1982). When this algorithm is optimized, there are about 25 percent fewer multiplications and slightly fewer additions than in a radix-2 algorithm. The algorithm has the disadvantage of only being applicable to data sets where the number of points is a power of four up to a maximum of 4,096 points whether in Turbo-87 or standard Turbo. A reduction in execution time of about 20 percent is accomplished when FFTB4.INC or FFT87B4.INC is used over its B2 counterpart.

FFTB4.INC performs complex multiplication with three real multiplications and three real additions and thus is most appropriate when multiplications take much more time than additions.

FFT87B4.INC performs complex multiplication with four real multiplications and two additions and thus is most appropriate with a numeric coprocessor.

Table 10-1 shows the recommended use of the four FFT algorithms for optimal performance.

Table 10-1 Four Fast Fourier Transforms

Number of Points	Without Coprocessor	With Coprocessor
2^k	FFT87B2.INC	FFT87B2.INC
4^k	FFTB4.INC	FFT87B4.INC

Although each of the algorithms is most efficient under different sets of circumstances, all four FFT algorithms will work whether you have a math coprocessor or not. The sample program defaults to the Turbo-87 radix-2 algorithm (FFT87B2.INC).

The Application Programs

There are six application programs that use the basic FFT routines contained in the previously mentioned include files (COMPFFT.INC, REALFFT.INC, COMPCNVL.INC, REALCNVL.INC, COMPCORR.INC, and REALCORR.INC).

Fast Fourier Transforms are particularly useful for analyzing periodic signals. Such a signal is represented by a function f satisfying

$$f(t + T) = f(t)$$

where t is time and T is the period. Under mild hypotheses, f can be expanded into a Fourier series such as the following:

$$f(t) = N^{-1/2} \sum_{n=-\infty}^{\infty} F(n) \exp(2\pi i n t/T)$$

where i is the square root of -1 . The term $\exp(2\pi i n t/T)$ is a sinusoid of period T/n and frequency n/T , and its coefficient $F(n)$ gives the strength of that frequency component in the original signal.

To analyze a signal on a digital computer, the signal must be *discretized*. Let $x(n)$ be computed by discretizing the function f at N equidistant points in one period. Thus, let

$$x(n) = f(nT/N) \quad n = 0, 1, \dots, N-1$$

Once we restrict attention to N points, it only makes sense to represent the signal in terms of N of the functions

$$\exp(2\pi i n t/T)$$

since the rest are redundant. For example:

$$\exp(2\pi i (-1) t/T) = \exp(2\pi i (N-1) t/T)$$

for $t = nT/N$, $n = 0, 1, \dots, N-1$. The Fourier series for the signal is then a finite sum, and has the form

$$x(n) = N^{-1/2} \sum_{k=0}^{N-1} X(k) \exp(2\pi i k n/N)$$

(The factor of $N^{-1/2}$ is a matter of convention. Some books do not include it in this formula, resulting in a factor of $1/N$ in the formula for X that follows.)

The formula for the coefficients $X(k)$ is as follows:

$$X(k) = N^{-1/2} \sum_{n=0}^{N-1} x(n) \exp(-i 2\pi n k/N)$$

This formula for X makes sense for any integer k . X is then periodic, satisfying

$$X(k + N) = X(k)$$

for all k . In formulas and programs, it is more convenient to let k run from 0 to $N-1$, but for analyzing signals it makes more sense to think of k as running from $(-N/2)$ to $(N/2 - 1)$. This is because values of k near zero represent the low frequency information, and values of k near or greater than $N/2$ represent frequencies that are so high that the discretization is too coarse to realize them accurately anyway. Therefore, if k is between $N/2$ and N , $X(k)$ should be thought of as the coefficient of

$$\exp(2\pi i (k-N) t/T)$$

rather than

$$\exp(2\pi i k t/T)$$

In other words, negative frequencies are represented on the right half of the transform.

COMPFFT.INC simply takes the complex Fast Fourier Transform of a set of complex data points. The complex Fourier transform is defined as

$$X_f = N^{-1/2} \sum_{n=0}^{N-1} x_n \exp(-2\pi i f n/N) \quad f = 0..N-1$$

where i is the square root of -1 . The inverse Fourier transform (which may also be calculated with COMPFFT.INC) is defined as

$$\bar{x}_n = N^{-1/2} \sum_{f=0}^{N-1} \bar{X}_f \exp(-2\pi i f n/N) \quad n = 0..N-1$$

where the bar stands for complex conjugation.

REALFFT.INC provides a procedure that is optimized for a discrete Fourier transform with all real data. It proceeds by mapping the N real data points onto $N/2$ complex points, applying one of the FFT routines, then reconstructing the N points of the desired transform. This reduces the computation time by about 25 percent compared to applying the complex FFT routine to the N real data points. REALFFT.INC can be used with any of the given FFT methods, but note that if a radix-4 method is used (FFTB4.INC or FFT87B4.INC), $N/2$ must be a power of four; so N must be of the form $2 * 4^k$.

COMPCNVL.INC provides a procedure for calculating convolutions of two complex vectors (Brigham 1974; Nussbaumer 1982). The discrete convolution of two complex functions x and h is defined by

$$y_m = \sum_{n=0}^{N-1} x_n h_{m-n} \quad m = 0, 1, \dots, N-1$$

where subscripts are taken *modulo* N (*circular convolution*). The basic theorem that allows us to calculate these effectively using FFTs is shown in the following:

$$Y_m = X_m H_m \quad m = 0, 1, \dots, N-1$$

where capital letters indicate the transforms of the functions represented by lower-case letters. Thus the procedure for convolution works like this:

1. Transform both given data sets using FFTs.
2. Multiply the resulting transforms point by point.
3. Find the inverse transform of this product using FFTs.

REALCNVL.INC provides a procedure for calculating convolutions of two real vectors (Brigham 1984; Nussbaumer 1982). This procedure is exactly the same as the previous procedure (COMPCNVL.INC) for complex convolution except that only one forward Fourier transform need be performed. The procedure is as follows:

1. Given two real vectors X_{Real} and H_{Real} , combine them into a complex vector X_{Real} plus iH_{Real} , where i is the square root of -1 .
2. Transform this complex vector.
3. Extract the transforms of the two real functions from the transform of the complex function (using the symmetry $X_f = \bar{X}_{-f}$, where the bar stands for complex conjugation).
4. Multiply the resulting transforms point by point.
5. Find the inverse transform of this product using FFTs. REALCNVL.INC is about 25 percent faster than its complex counterpart for the same set of real data.

COMPCORR.INC provides a procedure for calculating the crosscorrelation of two discrete complex functions or the autocorrelation of one discrete complex function (Brigham 1974). If x and h are the given discrete functions, then their correlation is defined as

$$c_m = \sum_{n=0}^{N-1} x_n h_{n+m} \quad m = 0, 1, \dots, N-1$$

where subscripts are taken modulo N (circular convolution). This can be computed using FFTs with a method analogous to that used in COMPCNVL.INC:

$$C_m = X_m H_{N-m} \quad m = 0, 1, \dots, N-1$$

Commonly x and h are real functions; in which case the preceding formula reduces to $C_m = X_m \bar{H}_m$, where the bar stands for complex conjugation. Thus the procedure for correlation works like this:

1. Transform both given data sets using FFTs.
2. Multiply each element of the transform of the first data set by the appropriate element of the transform of the second data.
3. Find the inverse transform of this product using FFTs.

REALCORR.INC provides a procedure for calculating the crosscorrelation of two discrete real functions or the autocorrelation of one discrete real function (Brigham 1974). This procedure is exactly the same as the previous procedure for

complex correlation except that only one forward Fourier transform need be performed. The procedure is as follows:

1. Given two real vectors X_{Real} and H_{Real} , combine them into a complex vector $X_{Real} + iH_{Real}$, where i is the square root of -1 .
2. Transform this complex vector.
3. Extract the transforms of the two real vectors from the transform of the complex vector (using the symmetry $X_f = \overline{X_{-f}}$, where the bar stands for complex conjugation).
4. Multiply each element of the transform of the first data set by the appropriate element of the transform of the second data.
5. Find the inverse transform of this product using FFTs.

Any one of the FFT include files can be used with any of the applications.

Data Sampling

While sampling theory is beyond the scope of this Toolbox, we would like to mention several common problems associated with data sampling (Brigham 1974; Press et al. 1986, Ch.12). The most common frustration is *aliasing*. A Fourier transform only represents frequencies up to a certain limit (called the *Nyquist limit*, or *Nyquist frequency*), which is given by half the sampling rate. (For example, if a signal is sampled sixty times a second, the Nyquist frequency will be 30 Hz.) A sample containing frequencies greater than this limit will not be properly transformed. The high frequencies will falsely contribute to the transform. This contribution will be indistinguishable from a contribution of a frequency below the Nyquist frequency.

There are several ways to combat aliasing. Increasing the sampling rate will increase the Nyquist frequency and thus reduce aliasing effects. It is also possible to pass the signal through a low pass filter, thus eliminating the high frequencies before sampling. If the Fourier transform of a signal does not converge to zero at the Nyquist frequency, the transform has very likely been aliased.

The Fourier transform assumes that the sample represents a periodic function and that the sample is an integer multiple of one period. If the latter condition is not true, spurious frequencies will show up in the transform. For example, if a sine wave is sampled from 0 to 270 degrees (instead of the full period), a sharp boundary is created because the sine of 0 does not equal the sine of 270. High frequencies will be introduced into the transform to account for that sharp boundary.

The assumption of periodicity can cause problems when convolving or correlating two signals that are not periodic. The convolution of each point in a signal affects the points surrounding it (the nature and extent of the affect depends on the particular convolving function). The assumption of periodicity means that the convolution at one end of the signal will affect the other end of the signal. This “end effect” can be eliminated by padding the data (on either end) with a sufficient number of zeros.

User-Defined Types

```
TNvector = array[0..TNArraySize] of Real;
```

```
TNvectorPtr = ^TNvector;
```

These user-defined types are different from others in this Toolbox, because they involve pointers. Pointers are used to transcend the limitations imposed by the 64K data segment size of Turbo Pascal. One array of 8,000 elements uses the entire data segment (in Turbo-87). However, it is possible to store these arrays on the heap, and to point to them with pointers that only require 4 bytes. The size of the heap (and hence the maximum size and number of *TNvectors*) is determined by the amount of memory in the machine.

Fast Fourier Transform Algorithms

The following documentation generally applies to all four FFT algorithms (FFTB2.INC, FFTB4.INC, FFT87B2.INC, FFT87B4.INC). When a difference between the radix-2 and radix-4 algorithms needs to be described, the radix-4 information will be placed in brackets following the radix-2 information (for example, the number of points must be a power of two [four]). The following describes the three procedures contained in each of the include files.

Procedure TestInput

Description

This example determines the number of data points in terms of a power of two [four]. If the number of data points is not a power of two [four], then an error is returned.

Input Parameters

NumPoints : Integer; Number of data points

The preceding parameter must satisfy the following conditions:

1. *NumPoints* ≥ 2 .
2. *NumPoints* must be a power of two [four].

Output Parameters

NumberOfBits : Byte; Number of data points as a power of two [four]

Error : Byte; 0: No errors
 1: *NumPoints* < 2
 2: *NumPoints* not a power of two [four]

Syntax of the Procedure Call

TestInput(NumPoints, NumberOfBits, Error);

Procedure MakeSinCosTable

Description

This example creates a look-up table of $\text{NumPoints}/2$ [3/4 NumPoints] roots of unity. The roots of unity are defined as follows:

$$\text{Root}_n = \exp(-i 2\pi n/\text{NumPoints}) \quad n = 0..\text{NumPoints}/2 \text{ [3/4 NumPoints]}$$

where i is the square root of -1 . These values are stored in two tables: *SinTable*, containing the imaginary parts of the roots of unity, and *CosTable*, containing the real parts of the roots of unity. It is faster to look up these values in a table than to calculate them in the FFT procedure.

Input Parameters

`NumPoints` : Integer; Number of data points

The preceding parameter must satisfy the following conditions:

1. $\text{NumPoints} \geq 2$.
2. NumPoints must be a power of two [four].

Output Parameters

`SinTable` : `TNvectorPtr`; Table of sine values

`CosTable` : `TNvectorPtr`; Table of cosine values

Syntax of the Procedure Call

`MakeSinCosTable(NumPoints, SinTable, CosTable);`

Procedure FFT

Description

This example implements the particular variation of the Cooley-Tukey algorithm. The Fast Fourier Transform of the data *XReal*, *XImag* is made in place and is thus returned in the vectors *XReal*, *XImag*. The inverse transform of the data can also be calculated with this procedure.

It is essential that procedures *TestInput* and *MakeSinCosTable* be called before procedure *FFT* is called. *TestInput* will flag any errors in the data (for example, number of points that are not a power of two [four]), and *MakeSinCosTable* generates a table of sine and cosine values referenced by *FFT*. *TestInput* and *MakeSinCosTable* need only be called once, even if several calls to *FFT* are made within the same program (for example, when computing the discrete convolution), as long as the number of data points is unchanged. If the number of data points changes between two calls of *FFT*, *TestInput* and *MakeSinCosTable* must be called again. (Interested readers are urged to consult the references given in the beginning of the chapter for details about the Cooley-Tukey algorithm.)

Input Parameters

NumberOfBits : Byte;	Number of data points as a power of two [four]
NumPoints : Integer;	Number of data points
Inverse : Boolean;	FALSE equals forward transform; TRUE equals inverse transform
XReal : TNvectorPtr;	Pointer to real values of the data points
XImag : TNvectorPtr;	Pointer to imaginary values of the data points
SinTable : TNvectorPtr;	Table of sine values
CosTable : TNvectorPtr;	Table of cosine values

The preceding parameters must satisfy the following conditions:

1. *NumPoints* ≥ 2 .
2. *NumPoints* must be a power of two [four].

Output Parameters

XReal : TNvectorPtr;	Pointer to real values of the discrete Fourier transform of the input data
XImag : TNvectorPtr;	Pointer to imaginary values of the discrete Fourier transform of the input data

Syntax of the Procedure Call

```
FFT(NumberOfBits, NumPoints, Inverse, XReal, XImag, SinTable, CosTable);
```

Fast Fourier Transform Applications

Each of the six application programs calls the three procedures contained within FFT algorithm files.

COMPFFT.INC

Description

This example is the most basic application, performing a complex Fast Fourier Transform. It simply calls *TestInput*, *MakeSinCosTable*, and *FFT* sequentially; thus accomplishing an in-place transformation of the complex data *XReal*, *XImag*.

Input Parameters

NumPoints : Integer; Number of data points

Inverse : Boolean; FALSE equals forward transform; TRUE equals inverse transform

XReal : TVectorPtr; Pointer to real values of the data points

XImag : TVectorPtr; Pointer to imaginary values of the data points

The preceding parameters must satisfy the following conditions:

1. *NumPoints* ≥ 2 .
2. *NumPoints* must be a power of two [four].

Output Parameters

XReal : TVectorPtr; Pointer to real values of the discrete Fourier transform of the input data

XImag : TVectorPtr; Pointer to imaginary values of the discrete Fourier transform of the input data

Error : Byte;
0: No errors
1: *NumPoints* < 2
2: *NumPoints* not a power of two [four]

Syntax of the Procedure Call

```
ComplexFFT(NumPoints, Inverse, XReal, XImag, Error);
```

Comments

The complex Fast Fourier method uses the *New/Dispose* procedures to manipulate the heap and should not be used in any program that uses *Mark/Release* to manipulate the heap.

REALFFT.INC

Description

This example performs a complex Fast Fourier Transform of real data. The *NumPoints* real data points are first mapped onto *NumPoints/2* complex data points. A complex Fast Fourier Transform of these complex points is performed by calling *TestInput*, *MakeSinCosTable*, and *FFT*. The *NumPoints/2* transform is then mapped onto *NumPoints* complex points. The real part of the transformation will be even, and the imaginary part of the transformation will be odd. If you are implementing this application with a radix-4 algorithm, be sure that the number of real data points (*NumPoints*) is twice the power of four.

Input Parameters

NumPoints : Integer; Number of data points

Inverse : Boolean; FALSE equals forward transform; TRUE equals inverse transform

XReal : TNvectorPtr; Pointer to real values of the data points

The preceding parameters must satisfy the following conditions:

1. *NumPoints* ≥ 4 .
2. *NumPoints* must be a power of two (twice a power of four for a radix-4 algorithm).

At least four data points are required, because this algorithm transforms the real vector to a complex vector half the size. If only two real data points were entered, the routine would have to take the transform of a single complex point.

Output Parameters

XReal : TNvectorPtr; Pointer to real values of the Fourier transform of the input data
XImag : TNvectorPtr; Pointer to imaginary values of the Fourier transform of the input data

Error : Byte; 0: No errors
 1: *NumPoints* < 4
 2: *NumPoints* not a power of two [twice a power of four]

Syntax of the Procedure Call

RealFFT(NumPoints, Inverse, XReal, XImag, Error);

Comments

This method uses the *New/Dispose* procedures to manipulate the heap and should not be used in any program that uses *Mark/Release* to manipulate the heap.

COMPCNVLINC

Description

The calculation of the convolution of two complex vectors is facilitated with a Fast Fourier Transform routine. The discrete convolution of two functions x and h is defined by

$$y_m = \sum_{n=0}^{N-1} x_n h_{m-n} \quad m = 0, 1, \dots, N-1$$

where subscripts are taken modulo N (circular convolution). The basic theorem that allows us to calculate these effectively using FFTs is as follows:

$$Y_m = X_m H_m \quad m = 0, 1, \dots, N-1$$

where capital letters indicate the transforms of the functions represented by lower-case letters. Thus the procedure for convolution works like this:

1. Transform both given data sets using FFTs.
2. Multiply the resulting transforms point by point.
3. Find the inverse transform of this product using FFTs.

Input Parameters

NumPoints : Integer; Number of data points

XReal : TNvectorPtr; Pointer to real values of the first set of data points

XImag : TNvectorPtr; Pointer to imaginary values of the first set of data points

HReal : TNvectorPtr; Pointer to real values of the second set of data points

HImag : TNvectorPtr; Pointer to imaginary values of the second set of data points

The preceding parameters must satisfy the following conditions:

1. *NumPoints* ≥ 2 .
2. *NumPoints* must be a power of two [four].

Output Parameters

XReal : TNvectorPtr; Pointer to real values of the convolution of *XReal*, *XImag* and *HReal*, *HImag*

XImag : TNvectorPtr; Pointer to imaginary values of the convolution of *XReal*, *XImag* and *HReal*, *HImag*

Error : Byte; 0: No errors
 1: *NumPoints* < 2
 2: *NumPoints* not a power of two [four]

Syntax of the Procedure Call

ComplexConvolution(NumPoints, XReal, XImag, HReal, HImag, Error);

Comments

This method uses the *New/Dispose* procedures to manipulate the heap and should not be used in any program that uses *Mark/Release* to manipulate the heap.

Description

The calculation of the convolution of two real vectors is facilitated with a Fast Fourier Transform routine. This procedure is exactly the same as the previous procedure for complex convolution (COMPCNVL.INC) except that only one Fourier transform need be performed. The procedure is as follows:

1. Given two real vectors $XReal$ and $HReal$, combine them into a complex vector $XReal + iHReal$, where i is the square root of -1 .
2. Transform this complex vector.
3. Extract the transforms of the two real functions from the transform of the complex function (using the symmetry $X_f = \bar{X}_{-f}$, where the bar stands for complex conjugation).
4. Multiply the resulting transforms point by point.
5. Find the inverse transform of this product using FFTs. REALCNVL.INC is about 25 percent faster than its complex counterpart for the same set of real data.

Input Parameters

NumPoints : Integer; Number of data points

XReal : TNvectorPtr; Pointer to real values of the first set of data points

HReal : TNvectorPtr; Pointer to real values of the second set of data points

The preceding parameters must satisfy the following conditions:

1. $NumPoints \geq 2$.
2. $NumPoints$ must be a power of two [four].

Output Parameters

XReal : TNvectorPtr; Pointer to real values of the convolution of $XReal$ and $HReal$

XImag : TNvectorPtr; Pointer to imaginary values of the convolution of $XReal$ and $HReal$

Error : Byte; 0: No errors

1: $NumPoints < 2$

2: $NumPoints$ not a power of two [four]

Syntax of the Procedure Call

```
RealConvolution(NumPoints, XReal, XImag, HReal, Error);
```

Comments

This method uses the *New/Dispose* procedures to manipulate the heap and should not be used in any program that uses *Mark/Release* to manipulate the heap.

COMPCORR.INC

Description

The calculation of the correlation of two complex vectors is facilitated with a Fast Fourier Transform routine. The discrete correlation of two complex functions x and h is defined by

$$y_m = \sum_{n=0}^{N-1} x_n h_{m+n} \quad m = 0, 1, \dots, N-1$$

where subscripts are taken modulo N (circular correlation). The basic theorem that allows us to calculate these effectively using FFTs is as follows:

$$Y_m = X_m H_{N-m} \quad m = 0, 1, \dots, N-1$$

where capital letters indicate the transforms of the functions represented by lower-case letters and $-$ indicates the complex conjugate. (Commonly x and h are real functions, in which case the preceding formula reduces to the more familiar expression $C_m = X_m \overline{H}_m$, where the bar stands for complex conjugation. (See REAL CORR.INC for a real version of correlation.) Thus the procedure for correlation works like this:

1. Transform both given data sets using FFTs.
2. Multiply each element of the transform of the first data set by the appropriate element of the transform of the second data.
3. Find the inverse transform of this product using FFTs.

If the functions x and h are different, the correlation is called *crosscorrelation*; if the functions x and h are the same, the correlation is called *autocorrelation*.

Input Parameters

NumPoints : Integer; Number of data points

Auto : Boolean; FALSE equals crosscorrelation; TRUE equals autocorrelation

XReal : TNvectorPtr; Pointer to real values of the first set of data points

XImag : TNvectorPtr; Pointer to imaginary values of the first set of data points

HReal : TNvectorPtr; Pointer to real values of the second set of data points (for cross-correlation)

HImag : TNvectorPtr; Pointer to imaginary values of the second set of data points (for crosscorrelation)

The preceding parameters must satisfy the following conditions:

1. *NumPoints* ≥ 2 .
2. *NumPoints* must be a power of two [four].

Output Parameters

XReal : TNvectorPtr; Pointer to real values of the correlation of *XReal*, *XImag* and *HReal*, *HImag* (or the autocorrelation of *XReal*, *XImag* if *Auto* = TRUE)

XImag : TNvectorPtr; Pointer to imaginary values of the correlation of *XReal*, *XImag* and *HReal*, *HImag* (or the autocorrelation of *XReal*, *XImag* if *Auto* = TRUE)

Error : Byte; 0: No errors
1: *NumPoints* < 2
2: *NumPoints* not a power of two [four]

Syntax of the Procedure Call

ComplexCorrelation(NumPoints, Auto, XReal, XImag, HReal, HImag, Error);

Comments

If you are performing an autocorrelation of the vector *XReal*, *XImag*, then set *Auto* = TRUE. In this case, the vector *HReal*, *HImag* will not contain any information but must still be passed into the procedure. Autocorrelations are faster to compute, since only one forward transformation must be made, as opposed to two for crosscorrelation.

This method uses the *New/Dispose* procedures to manipulate the heap and should not be used in any program that uses *Mark/Release* to manipulate the heap.

REALCORR.INC

Description

The calculation of the convolution of two real vectors is facilitated with a Fast Fourier Transform routine. This procedure is exactly the same as the previous procedure for complex correlation (COMPCORR.INC) except that only one forward Fourier transform need be performed. The procedure is as follows:

1. Given two real vectors *XReal* and *HReal*, combine them into a complex vector $XReal + iHReal$, where i is the square root of -1 .
2. Transform this complex vector.
3. Extract the transforms of the two real vectors from the transform of the complex vector (using the symmetry $X_f = \bar{X}_{-f}$, where the bar stands for complex conjugation).
4. Multiply each element of the transform of the first data set by the appropriate element of the transform of the second data.
5. Find the inverse transform of this product using FFTs.

Input Parameters

NumPoints : Integer; Number of data points

Auto : Boolean; FALSE equals crosscorrelation; TRUE equals autocorrelation

XReal : TNvectorPtr; Pointer to real values of the first set of data points

HReal : TNvectorPtr; Pointer to real values of the second set of data points (for cross-correlation)

The preceding parameters must satisfy the following conditions:

1. *NumPoints* ≥ 2 .
2. *NumPoints* must be a power of two [four].

Output Parameters

XReal : TNvectorPtr; Pointer to real values of the correlation of *XReal* and *HReal* (or the autocorrelation of *XReal* if *Auto* = TRUE)
XImag : TNvectorPtr; Pointer to imaginary values of the correlation of *XReal* and *HReal* (or the autocorrelation of *XReal* if *Auto* = TRUE)
Error : Byte; 0: No errors
 1: *NumPoints* < 2
 2: *NumPoints* not a power of two [four]

Syntax of the Procedure Call

RealCorrelation(NumPoints, Auto, XReal, XImag, HReal, Error);

Comments

If you are performing an autocorrelation of the vector *XReal*, then set *Auto* equal to TRUE. In this case, the vector *HReal* will not contain any information but must still be passed into the procedure. Autocorrelations are faster to compute, since only one forward transformation must be made, as opposed to two for crosscorrelation. This method uses the *New/Dispose* procedures to manipulate the heap and should not be used in any program that uses *Mark/Release* to manipulate the heap.

Sample Program

The sample program FFTPROGS.PAS provides I/O functions that demonstrate any of the application programs. The FFT algorithm routines are included with the following statements:

```
(*{$I FFTB2.INC}      (* radix 2, 8088 version *)
(*{$I FFTB4.INC}      (* radix 4, 8088 version *)
{$I FFT87B2.INC}      (* radix 2, 8087 version *)
(*{$I FFT87B4.INC}    (* radix 4, 8087 version *)
```

As you can see, three of the include statements must be commented-off so that only one file is included. To change which file is included, simply comment-off the one that is currently active (in this example that would be FFT87B2.INC) and remove the comment symbol (*) from the include file of your choice.

Input File

Data may be entered from a text file. The real and imaginary parts of a complex number should be separated by a space and followed by a carriage return. Real numbers should each be followed by a carriage return.

The application files COMPFFT.INC, COMPCNVL.INC, and COMP CORR.INC expect data to be in complex form. A data file containing a four-point complex square wave could look like this:

```
0 0
1 1
1 1
0 0
```

The application files REALFFT.INC, REALCNVL.INC, and REAL CORR.INC expect data to be in real form. A data file containing a four-point real square wave could look like this:

```
0
1
1
0
```

Example

Problem. Perform a Fourier transform and an autocorrelation of a 32-point square wave. Also, convolve and crosscorrelate this square wave with a saw-tooth wave (assume you are working in Turbo-87).

1. First, make sure that the FFT file FFT87B2.INC is included in FFTPROGS.PAS:

```
(*{$I FFTB2.INC}      (* radix 2, 8088 version *)
(*{$I FFTB4.INC}      (* radix 4, 8088 version *)
{$I FFT87B2.INC}      (* radix 2, 8087 version *)
(*{$I FFT87B4.INC}    (* radix 4, 8087 version *)
```

The input data file SAMP10A.DAT is as follows (note that this is in real format):

```
0
0
0
0
0
0
0
0
0
```

0
0
0
1
1
1
1
1
1
1
1
1
1
1
1
1
0
0
0
0
0
0
0
0
0
0
0

2. Run FFTPROGS.PAS:

1. Real Fast Fourier Transform
2. Real Convolution
3. Real Autocorrelation
4. Real Crosscorrelation
5. Complex Fast Fourier Transform
6. Complex Convolution
7. Complex Autocorrelation
8. Complex Crosscorrelation

Select a number (1-8): 1

***** Real Fast Fourier Transform *****

(F)orward or (I)nverse transform? F

Enter data from (K)eyboard or (F)ile? F

File name? SAMP10A.DAT

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

Results of Real Fourier Transform:

1.94454364826301E+000	0.00000000000000E+000
-1.59057003804788E+000	-3.14018491736755E-016
7.53417436515731E-001	5.88784672006416E-017
5.96901852132470E-002	-2.94392336003208E-016
-4.26776695296637E-001	3.92523114670944E-017
2.89883706652938E-001	-2.94392336003208E-016
6.20757203331860E-002	7.85046229341887E-017
-2.66655959906343E-001	-5.39719282672548E-017
1.76776695296637E-001	0.00000000000000E+000
6.63840517512576E-002	2.06074635202245E-016
-2.08522329739913E-001	1.76635401601925E-016
1.27160952826887E-001	-1.17756934401283E-016
7.32233047033632E-002	-1.37383090134830E-016
-1.83841625879619E-001	-6.86915450674151E-017
1.00135954077543E-001	2.74766180269661E-016
8.37351650164209E-002	1.37383090134830E-016
-1.76776695296637E-001	0.00000000000000E+000
8.37351650164209E-002	-1.37383090134830E-016
1.00135954077543E-001	-2.74766180269661E-016
-1.83841625879619E-001	6.86915450674151E-017
7.32233047033632E-002	1.37383090134830E-016
1.27160952826887E-001	1.17756934401283E-016
-2.08522329739913E-001	-1.76635401601925E-016
6.63840517512576E-002	-2.06074635202245E-016
1.76776695296637E-001	0.00000000000000E+000
-2.66655959906343E-001	5.39719282672548E-017
6.20757203331860E-002	-7.85046229341887E-017
2.89883706652938E-001	2.94392336003208E-016
-4.26776695296637E-001	-3.92523114670944E-017
5.96901852132470E-002	2.94392336003208E-016
7.53417436515731E-001	-5.88784672006416E-017
-1.59057003804788E+000	3.14018491736755E-016

Note that the transform of the even real-square wave is an even real function. If you take the inverse transform of this data, you should get back the original square wave.

3. Run FFTPROGS.PAS:

1. Real Fast Fourier Transform
2. Real Convolution
3. Real Autocorrelation
4. Real Crosscorrelation
5. Complex Fast Fourier Transform
6. Complex Convolution
7. Complex Autocorrelation
8. Complex Crosscorrelation

Select a number (1-8): 5

***** Complex Fast Fourier Transform *****

(F)orward or (I)nverse transform? I

Enter data from (K)eyboard or (F)ile? F

File name? SAMP10B.DAT

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

Results of Complex Fourier Transform:

```
1.88411095042053E-015  0.00000000000000E+000
1.72710170455215E-015  0.00000000000000E+000
2.04112019628891E-015  -3.92523114670944E-017
2.04112019628891E-015  -1.19825492681322E-016
1.17756934401283E-015  0.00000000000000E+000
1.09906472107864E-015  -3.14264671746203E-016
1.02325930356393E-015  1.53635757788210E-016
1.02243303999922E-015  -3.53529506016654E-017
1.07086707894778E-015  0.00000000000000E+000
3.60019188480208E-016  2.56022496768081E-016
-3.89233270203019E-017  1.57501605887274E-016
1.00000000000000E+000  4.33843984418077E-016
1.00000000000000E+000  0.00000000000000E+000
1.00000000000000E+000  4.71273917614581E-016
1.00000000000000E+000  2.94392336003208E-016
1.00000000000000E+000  5.69158516272868E-016
9.99999999999999E-001  0.00000000000000E+000
1.00000000000000E+000  3.92523114670944E-017
1.00000000000000E+000  3.92523114670944E-017
9.99999999999999E-001  -4.13208697471334E-017
1.00000000000000E+000  0.00000000000000E+000
1.00000000000000E+000  1.56763065858929E-016
-8.12038283536641E-017  -1.53635757788210E-016
2.33640926947802E-016  -2.78665541135090E-016
9.70253117341126E-016  0.00000000000000E+000
1.21007327020357E-015  -2.95274808235175E-016
4.31446441691246E-016  -1.57501605887274E-016
1.80560632748634E-015  -2.72697621989622E-016
9.42055475210265E-016  0.00000000000000E+000
2.04112019628891E-015  -3.13772311727307E-016
1.80560632748634E-015  -2.94392336003208E-016
1.96261557335472E-015  -2.55140024536113E-016
```

You get back the original square wave, accurate to 15 significant figures.

The autocorrelation of a square wave is simply a triangle. Let's take the autocorrelation of the square wave.

4. Run FFTPROGS.PAS:

1. Real Fast Fourier Transform
2. Real Convolution
3. Real Autocorrelation
4. Real Crosscorrelation
5. Complex Fast Fourier Transform
6. Complex Convolution
7. Complex Autocorrelation
8. Complex Crosscorrelation

Select a number (1-8): 3

***** Real Autocorrelation *****

Enter data from (K)eyboard or (F)ile? F

File name? SAMP10A.DAT

Direct output to one of the following:

(S)creen
(P)rinter
(F)ile

Results of Autocorrelation:

1.94454364826301E+000	-6.89287897017933E-016
1.76776695296637E+000	-5.49532360539321E-016
1.59099025766973E+000	-6.55792559089139E-016
1.41421356237309E+000	-4.17055809337878E-016
1.23743686707646E+000	-4.44089209850063E-016
1.06066017177982E+000	-2.74766180269661E-016
8.83883476483184E-001	-3.84393694788862E-016
7.07106781186547E-001	-1.17756934401283E-016
5.30330085889910E-001	5.37799391805346E-017
3.53553390593274E-001	-1.96261557335472E-016
1.76776695296637E-001	-1.88132137453390E-016
0.00000000000000E+000	-9.32242397343491E-017
-3.92523114670944E-016	4.44089209850063E-016
2.35513868802566E-016	0.00000000000000E+000
4.71027737605132E-016	-1.06260198549818E-016
4.71027737605132E-016	-1.66822323735151E-016
1.57009245868377E-016	5.81728018656864E-016
-3.14018491736755E-016	3.92523114670944E-016
0.00000000000000E+000	6.00281407857881E-016
-1.57009245868377E-016	2.89485797069821E-016
-3.92523114670944E-016	4.44089209850063E-016
-3.14018491736755E-016	1.17756934401283E-016
1.76776695296636E-001	4.39904846020120E-016
3.53553390593273E-001	-1.96261557335472E-017
5.30330085889910E-001	5.37799391805346E-017
7.07106781186547E-001	3.53270803203849E-016
8.83883476483184E-001	2.43643288684648E-016
1.06066017177982E+000	2.20794252002406E-016
1.23743686707646E+000	-4.44089209850063E-016
1.41421356237309E+000	1.57009245868377E-016
1.59099025766973E+000	5.07490473185598E-017
1.76776695296637E+000	3.04205413869981E-016

Keeping in mind that this is a periodic function (see "Data Sampling"), you can see that this is a triangle wave.

Let's now convolve the square wave with a saw-tooth wave. The input file for the saw-tooth wave (SAMP10C.DAT) is as follows:

0
0
0
0
0
0

0
0
0
0
0
1
2
3
4
5
6
7
8
9
10
1
1
0
0
0
0
0
0
0
0
0
0
0

5. Run FFTPROGS.PAS:

1. Real Fast Fourier Transform
2. Real Convolution
3. Real Autocorrelation
4. Real Crosscorrelation
5. Complex Fast Fourier Transform
6. Complex Convolution
7. Complex Autocorrelation
8. Complex Crosscorrelation

Select a number (1-8): 2

***** Real Convolution *****

Enter data from (K)eyboard or (F)ile? F

The first function:

File name? SAMP10A.DAT

The second function:

File name? SAMP10C.DAT

Direct output to one of the following:

(S)screen

(P)printer

(F)file

Results of Real Convolution:

1.16672618895780E+001	0.00000000000000E+000
1.14904851942814E+001	1.09906472107864E-015
1.11369318036881E+001	6.28036983473510E-016
1.06066017177982E+001	4.71027737605132E-016
9.89949493661168E+000	0.00000000000000E+000
9.01561146012848E+000	7.85046229341887E-016
7.95495128834866E+000	-6.28036983473510E-016
6.71751442127220E+000	1.57009245868377E-016
5.30330085889910E+000	0.00000000000000E+000
3.71231060122936E+000	-1.96261557335472E-015
1.94454364826299E+000	-2.11962481922310E-015
-1.85270910124685E-014	-2.98317567149917E-015
-2.04112019628891E-014	0.00000000000000E+000
-1.13046657025232E-014	-3.29719416323593E-015
-1.44448506198907E-014	-3.45420340910430E-015
-9.42055475210265E-015	-3.45420340910430E-015
-1.50728876033642E-014	0.00000000000000E+000
-1.38168136364172E-014	-4.71027737605132E-016
-1.38168136364172E-014	-6.28036983473510E-016
-1.13046657025232E-014	-1.57009245868377E-016
-1.13046657025232E-014	0.00000000000000E+000
-5.02429586778808E-015	-1.09906472107864E-015
1.76776695296632E-001	6.28036983473510E-016
5.30330085889907E-001	-2.04112019628891E-015
1.06066017177982E+000	0.00000000000000E+000
1.76776695296637E+000	1.33457858988121E-015
2.65165042944956E+000	2.11962481922310E-015
3.71231060122938E+000	2.66915717976242E-015
4.94974746830584E+000	0.00000000000000E+000
6.36396103067893E+000	3.61121265497268E-015
7.95495128834867E+000	3.45420340910430E-015
9.72271824131504E+000	5.33831435952483E-015

Now let's crosscorrelate the square wave with the saw-tooth wave.

6. Run FFTPROGS.PAS:

1. Real Fast Fourier Transform
2. Real Convolution
3. Real Autocorrelation
4. Real Crosscorrelation
5. Complex Fast Fourier Transform
6. Complex Convolution
7. Complex Autocorrelation
8. Complex Crosscorrelation

Select a number (1-8): 4

***** Real Crosscorrelation *****

Enter data from (K)eyboard or (F)ile? F

The first function:

File name? SAMP10A.DAT

The second function:

File name? SAMP10C.DAT

Direct output to one of the following:

(S)creen

(P)rinter

(F)ile

Results of Real Crosscorrelation:

1.16672618895780E+001	0.00000000000000E+000
9.72271824131504E+000	-6.67289294940604E-016
7.95495128834866E+000	-7.85046229341887E-016
6.36396103067893E+000	-9.02803163743171E-016
4.94974746830583E+000	0.00000000000000E+000
3.71231060122937E+000	-1.49158783574959E-015
2.65165042944954E+000	-2.04112019628891E-015
1.76776695296636E+000	-1.72710170455215E-015
1.06066017177981E+000	0.00000000000000E+000
5.30330085889897E-001	-2.62990486829532E-015
1.76776695296621E-001	-3.29719416323593E-015
-1.38168136364172E-014	-2.86541873709789E-015
-1.63289615703113E-014	0.00000000000000E+000
-1.06766287190497E-014	-3.06168029443336E-015
-1.25607396694702E-014	-3.14018491736755E-015
-1.00485917355762E-014	-4.39625888431457E-015
-1.13046657025232E-014	0.00000000000000E+000
-9.42055475210265E-015	-3.53270803203849E-016
-5.02429586778808E-015	7.85046229341887E-016
-5.02429586778808E-015	-2.74766180269661E-016
-5.33831435952483E-015	0.00000000000000E+000
-3.76822190084106E-015	-5.49532360539321E-016
1.94454364826300E+000	2.04112019628891E-015
3.71231060122937E+000	4.71027737605132E-016
5.30330085889911E+000	0.00000000000000E+000
6.71751442127221E+000	3.65046496643978E-015
7.95495128834867E+000	3.29719416323593E-015
9.01561146012849E+000	4.04298808111072E-015
9.89949493661168E+000	0.00000000000000E+000
1.06066017177982E+001	5.10280049072227E-015
1.11369318036881E+001	3.14018491736755E-015
1.14904851942814E+001	5.65233285126159E-015

Graphics Programs

The programs in this chapter can only be run by those users with PC-DOS.

There are some programs that graphically demonstrate the usefulness of the least-squares routines in Chapter 9 and the Fourier transforms in Chapter 10. A graphics monitor is required. Each program reads a data set from an input file, and uses this Toolbox to display the results. You will see curves being fitted to data using the least-squares routines and also see a signal being transformed into its Fourier spectrum.

The programs LSQIBM.COM and FFTIBM.COM graphically illustrate the power and utility of the Turbo Pascal Numerical Methods Toolbox. To run them, you'll need an IBM Color Graphics Adapter (CGA) or a suitable clone. (The programs LSQHERC.COM AND FFTHERC.COM can be used to graphically illustrate the Toolbox on a Hercules Monochrome Graphics Adapter or compatible.) And to print, you'll need an Epson or IBM compatible dot-matrix printer. As explained in this chapter, these programs can be recompiled to run on other hardware, including the IBM Enhanced Graphics Adapter (EGA) in its high resolution mode. The programs can also be recompiled to take advantage of the 8087 (or 80287) math coprocessor.

Function of the Least-Squares Graphics Demonstration Program

The program LSQIBM.COM demonstrates the least-squares capabilities of the Toolbox. To run it, you must have

- 4x6.FON (Graphix fonts)
- 8x8.FON (Graphix fonts)
- ERROR.MSG (Graphix error messages)
- LSQIBM.COM
- SAMP11A.DAT (Input data file)

on the current directory. (The first three files are identical to those in the Graphix Toolbox.)

By default the input is a file called SAMP11A.DAT that has X and Y values (in ASCII form) separated by carriage returns. Running LSQIBM.COM will provide five different least-squares fits to the input data. The different fits are based on the function forms: logarithm, exponential, polynomial, power law, and finite Fourier series. The fits are displayed graphically on the screen and can be printed on an Epson or compatible printer.

The first plot shows the input data from SAMP11A.DAT along with three curves. The three curves are the graphs of the power function

$$Y = aX^b$$

the exponential function

$$Y = a \exp(bX)$$

and the logarithm function

$$Y = a \ln(bX)$$

The header to the plot tells which curve corresponds to which function. The next plot shows the same input data plotted with a five-term Fourier series:

$$Y = a + b * \cos(x) + c * \sin(x) + d * \cos(2X) + e * \sin(2X)$$

and a five-term polynomial (that is, a polynomial of degree four). The coefficients are found using the routines from Chapter 9, and they give the least-square error among all functions of that form. (In some cases, the problem is transformed into a linear problem, and the error is actually the least for the transformed problem but possibly not exactly the least for the original problem.) Again, the header to the plot tells which curve corresponds to which function.

Finally, a bar chart shows the error for each function. The data is not at all periodic, so the Fourier series model is the worst. The five-degree polynomial gives the best fit, but it is not much better than the fit obtained by using power, exponential, or logarithm functions.

Pressing the space bar allows you to cycle through the different screens. Pressing **Q** exits the program. Pressing **H** sends a hard copy to the printer (see the section entitled “Printing”).

You can use your own data to run the program by running LSQDEMO with two file names, such as

```
LSQIBM LSQIN.DAT LSQOUT.DAT
```

The input data from LSQIN.DAT along with the least-squares fits and coefficients are output to a text file called LSQOUT.DAT.

A default output data file can easily be arranged by changing a constant *WriteToFile* in LSQDEMO.PAS and recompiling it. (See the section, “Rebuilding LSQIBM.COM,” to recompile it, and the comment in LSQDEMO.PAS next to the constant *WriteToFile*.)

Function of the Fourier Transform Graphics Demonstration Program

The program FFTIBM.COM demonstrates the Fourier capabilities of the Toolbox. To run it, you must have

- 4x6.FON (Graphix fonts)
- 8x8.FON (Graphix fonts)
- ERROR.MSG (Graphix error messages)
- FFTIBM.COM
- SAMP11B.DAT (Input data file)

on the current directory. (The first three files are identical to those in the Graphix Toolbox.)

By default the input is a file called SAMP11B.DAT that has 1,024 real values (in ASCII form) separated by carriage returns. These represent sample points from a two-second signal sampled at a rate of 512 points per second. The program will display four FFT transforms at the following sampling rates: 8 per second (16 points), 32 per second (64 points), 128 per second (256 points), and 512 per second (1,024 points). For the last two samplings, the default data yields the same transforms, demonstrating that a sample rate higher than twice the highest frequency adds no new information (the *Nyquist limit*). The transforms are shown on a scale of -64 to $+63$ cycles per second.

In addition to the transforms, the program displays the inverse transform over the original data, illustrating the degree to which information is lost at different sampling rates. The header tells which curve is the original data and which is the inverse transform.

Pressing the space bar cycles through the four plots. To the right of each plot is two smaller plots that show the coefficients for the real and imaginary parts of the Fourier transform. Some information about the sample and the transform appears after each plot.

The graphs that appear on the screen can be printed on an Epson or compatible printer.

You can use your own data to run the program by running FFTDEMO with two file names on the command line, such as

```
FFTIBM FFTIN.DAT FFTOUT.DAT
```

The 1,024 complex points of the 1,024-point FFT will be written into the output file FFTOUT.DAT, with one complex number per line. The real and imaginary parts are separated by a space.

A default output data file can easily be arranged by changing the constant *WriteToFile* in FFTDEMO.PAS and recompiling it. (See the section, "Rebuilding FFTIBM.COM," to recompile it, and the comment in FFTDEMO.PAS next to *WriteToFile*.)

Printing

Both LSQIBM.COM and FFTIBM.COM provide the capability to print the graphs that appear on the screen. If you run these programs on the default data sets, the printouts will look just like the ones displayed here (see Figure 11-1, 11-2, and 11-3). This is particularly useful when you would like a permanent, visual record of your program results.

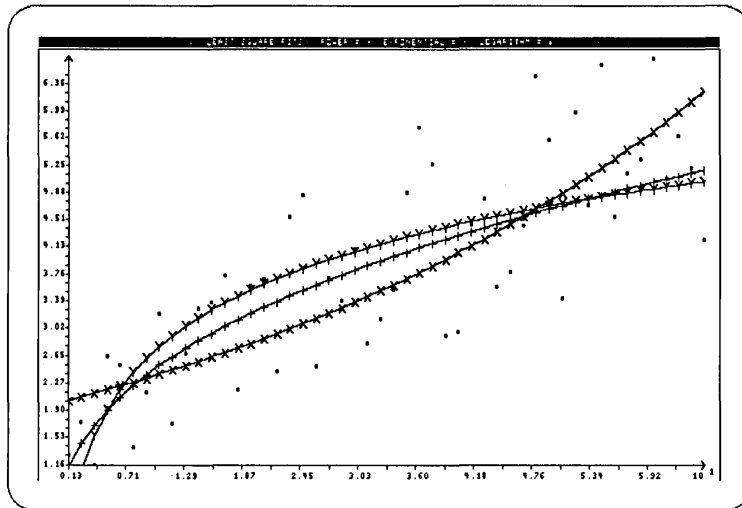


Figure 11-1 LSQIBM.COM

Least-square fits of power, exponential, and logarithmic functions to data in SAMP11A.DAT. Note: This was run without an 8087.

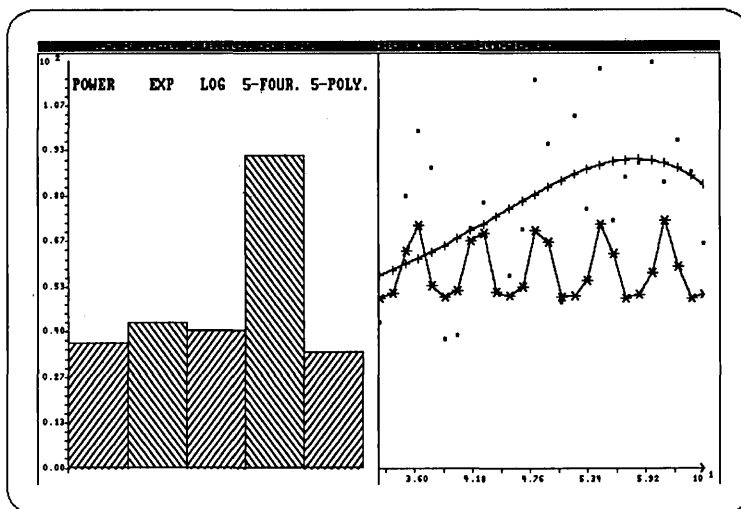


Figure 11-2 *LSQIBM.COM*

Sum of squares of residuals for five fits. Left side displays bar graph comparing various fits. Right side depicts same data fitted by 5th degree polynomial and a partial Fourier series. Note: This was run without an 8087.

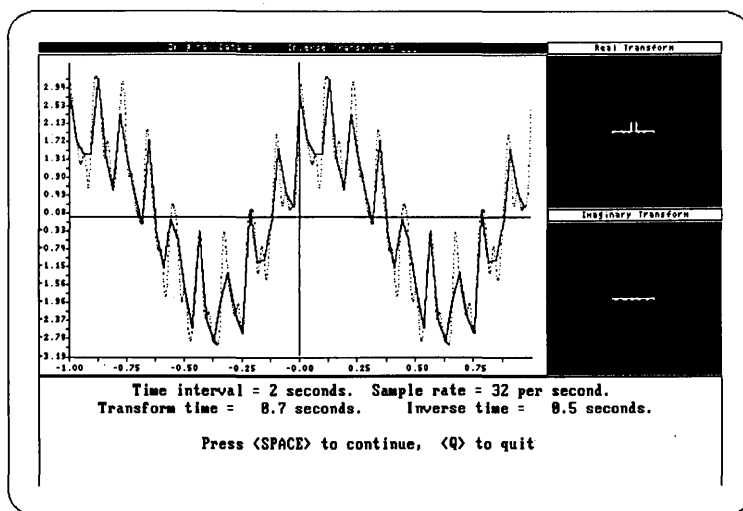


Figure 11-3 *FFTIBM.COM*

Graph depicts data of SAMP11B.DAT. Upper right-hand side displays real transform coefficients. Mid right-hand side displays imaginary coefficients. Dotted lines represent inverse transform of the Fourier transform, superimposed over original data. Inverse transform is not identical to original data because of coarseness of sample rate. Note: This was run without an 8087.

To get a printout, merely press **H** for **Hardcopy** at the menu prompt. An Epson or IBM compatible dot-matrix printer is required. (You cannot get a graphics printout on a daisy-wheel printer.)

If your printer is not Epson-compatible and the **Hardcopy** command is not functioning properly, you can rebuild the program with a different printer mode. (See the instructions that follow for rebuilding the program.) Both LSQDEMO.PAS and FFTDEMO.PAS have a constant called *PrintMode* near the beginning of the file. Setting *PrintMode* to the value 1 will allow printing on the largest number of printers, but a mode value of 4 will give the best looking output from a color display to an Epson printer. The default value is 6, since that gives good results for the Color Graphics Adapter, Hercules, and Enhanced Graphics Adapter.

You can also use the DOS program GRAPHICS.COM, which enables the **PrtSc** key to print the screen in graphics mode.

Rebuilding LSQIBM.COM

The sources to LSQIBM.COM are provided. To recompile, you need Turbo Pascal (version 3.0) and Turbo Graphix Toolbox (version 1.06A or later), as well as this Toolbox.

To rebuild LSQIBM.COM, the following files are needed:

From CHAP9:

EXP.LSQ (for the exponential model)
FOURIER.LSQ (Fourier series)
LOG.LSQ (logarithmic)
POLY.LSQ (polynomial)
POWER.LSQ (power law)

From CHAPII:

GENERIC.LSQ
IOCHECK.INC (a modification of COMMON.INC)
LEAST.MOD (a modification of LEAST.INC from Chapter 9)
LSQDEMO.PAS

From Turbo Pascal:

(These files are not included in this Toolbox.)

TURBO.COM or TURBO-87.COM (the Turbo Pascal compiler)
TURBO.MSG (compiler error messages)

From the Turbo Pascal Graphix Toolbox (version 1.06A or later):

(These files are not included in this Toolbox.)

AXIS.HGH

FINDWRLD.HGH

GRAPHIX.SYS (a copy of GRAPHIX.IBM)

HATCH.HGH

HISTOGRM.HGH

KERNEL.SYS

POLYGON.HGH

TYPEDEF.SYS

WINDOWS.SYS

Once you have all of these files on the current directory, enter Turbo Pascal and compile LSQDEMO.PAS to disk. The resulting LSQDEMO.COM should be renamed LSQIBM.COM to distinguish it from the Hercules version.

Rebuilding FFTIBM.COM

The sources to FFTIBM.COM are provided. To recompile, you need Turbo Pascal (version 3.0) and Turbo Graphix Toolbox (version 1.06A or later), as well as this Toolbox.

To rebuild FFTIBM.COM, the following files are needed:

From CHAPI0:

COMPFFT.INC

FFTB2.INC

REALFFT.INC

From CHAPII:

4X6.FON (originally from the Graphix Toolbox)

8X8.FON (originally from the Graphix Toolbox)

ERROR.MSG (originally from the Graphix Toolbox)

FFTDEMO.PAS

IOCHECK.INC (a modification of COMMON.INC)

From Turbo Pascal:

(These files are not included in this Toolbox.)

TURBO.COM or TURBO-87.COM (the Turbo Pascal compiler)

TURBO.MSG (compiler error messages)

From the Turbo Pascal Graphix Toolbox (version 1.06A or later):
(These files are not included in this Toolbox.)

AXIS.HGH

FINDWRLD.HGH

GRAPHIX.SYS (a copy of GRAPHIX.IBM)

KERNEL.SYS

POLYGON.HGH

TYPEDEF.SYS (with *MaxPlotGlb* changed to the value 1,024)

WINDOWS.SYS

The system constant *MaxPlotGlb* in the file TYPEDEF.SYS must be changed to have the value 1,024 instead of 100. Without this change, FFTDEMO will terminate with an error message.

Once you have all of these files on the current directory, enter Turbo Pascal and compile FFTDEMO.PAS to disk. The resulting FFTDEMO.COM should be renamed FFTIBM.COM to distinguish it from the Hercules version.

Rebuilding for the Hercules Card

To recompile for the Hercules graphics card, simply copy GRAPHIX.HGC to GRAPHIX.SYS, as explained in the Turbo Pascal Graphix Toolbox. The font file 14X9.FON is also required from the Graphix Toolbox, and the version of the Graphix Toolbox must be 1.07A or later. (You can use version 1.06A of the Graphix Toolbox if you're using the copy of GRAPHIX.HGC accompanying the other CHAPII files.)

The resulting files LSQDEMO.COM and FFTDEMO.COM should be renamed to LSQHERC.COM and FFTHERC.COM to distinguish them from the IBM CGA versions.

Rebuilding for the EGA Card

To recompile for the IBM Enhanced Graphics Adapter (EGA), you must copy GRAPHIX.EGA to GRAPHIX.SYS. A copy of GRAPHIX.EGA is included with this Toolbox since many users purchased copies of the Turbo Graphix Toolbox before support for the EGA was added.

LSQIBM.COM and FFTIBM.COM will run on machines with an EGA card. However, to take advantage of the higher resolution offered by this card, you must copy GRAPHIX.EGA and recompile.

Release 1.07A of the Graphics Toolbox also supports other graphics hardware. LSQDEMO and FFTDEMO can be recompiled for any of these systems.

Using the Math Coprocessor

To recompile to take advantage of the math coprocessor, you must use TURBO-87.COM instead of TURBO.COM. Some increased performance in the FFT demo program will be obtained if FFTB2.INC is replaced by FFT87B2.INC from Chapter 10 of the Toolbox.

References

Abramowitz, Milton, and Irene A. Stegun, eds. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Washington, D.C.: National Bureau of Standards Applied Mathematics Series, 55, 1972.

Atkinson, L.V., and P.J. Harley. *An Introduction to Numerical Methods with Pascal*. Reading: Addison-Wesley Publishing Co., 1983. This is an excellent text for learning numerical methods, with an emphasis on the implementation of various numerical algorithms.

Brigham, E. Oran. *The Fast Fourier Transform*. Englewood Cliffs: Prentice-Hall, Inc., 1974. A very complete, easy-to-read text on the use and implementation of the fast Fourier transform algorithm.

The next three texts are excellent for learning numerical analysis, emphasizing the mathematical theory underlying the algorithms in this toolbox.

Burden, Richard L., and J. Douglas Faires. *Numerical Analysis*, 3rd ed. Boston: Prindle, Weber & Schmidt, 1985.

Cheney, Ward, and David Kincaid. *Numerical Mathematics and Computing*, 2nd ed. Monterey: Brooks/Cole Publishing Co., 1985.

Dahlquist, Germund, and Ake Björck. *Numerical Methods*, trans. Ned Anderson. Englewood Cliffs: Prentice-Hall, Inc., 1974.

Fried, Stephen S. "Evaluating 8087 Performance on the IBM PC." *Byte*. Vol. 9, Number 9 (Special Issue), 1985, pp.197–208.

Gerald, Curtis F., and Patrick O. Wheatley. *Applied Numerical Analysis*, 3rd ed. Reading: Addison-Wesley Publishing Co., 1984. This is another excellent source for learning numerical analysis.

Press, William H., Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. New York: Cambridge University Press, 1986. This book is user-oriented, discusses many of the subtle problems encountered when implementing numerical methods, and has program listings in Turbo Pascal.

Ralston, Anthony, and Philip Rabinowitz. *A First Course in Numerical Analysis*. New York: McGraw-Hill Book Co., 1978. A well-written mathematics text that is a step more sophisticated than the preceding ones.

Index

\$FF error, 145, 190, 202

A

ADAMS_1.INC, 156, 168–170
ADAMS_1.PAS, 170–171
Adams-Bashforth/Adams-Moulton
predictor-corrector method, 156,
168–171
Adams-Bashforth formula, 168
Adams-Moulton formula, 168
ADAPGAUS.INC, 88, 98–100
ADAPGAUS.PAS, 101
ADAPSIMP.INC, 88, 95–96
ADAPSIMP.PAS, 96–97
Adaptive schemes, 87
quadrature, 95–101
Aliasing, 239

B

Backward substitution, 114, 120
Basis vectors, 222
BISECT.INC, 18–19
Bisection method, 15
root of a function using, 18–20
BISECT.PAS, 19–20
Boundary value problems, 155–156, 158
using Linear Shooting/Runge-Kutta
methods, 215–219
using Shooting/Runge-Kutta methods,
208–214

C

Chebyshev polynomials, 224
Color Graphics Adapter, 3, 12
for graphics demos, 261, 267
COMMON.INC, 7, 11, 13
COMPCNVL.INC, 233, 235, 237–238
application, 246–247
COMPCORR.INC, 233, 235, 238
application, 249–250
COMPFFT.INC, 233, 235, 237
application, 244–245
Compiler directives, 13
Convergence, rate of, 15–16
Cooley-Tukey method, 233–234
Coprocessor, 3, 270
CUBE_CLA.INC, 57–58
CUBE_CLA.PAS, 59–62
CUBE_FRE.INC, 52–53
CUBE_FRE.PAS, 53–56

Cubic spline methods
clamped, 44, 57–62
free, 44, 52–56, 64, 76–79
Cyclic Jacobi method, 132, 149–153

D

Data sampling, 239–240
Data types, 12
Defined constants, 12
Deflation, 16
and Laguerre, 37–41
and Newton-Horner, 28–32
of a matrix, 132, 143–148
DERIV2FN.INC, 64–65, 83–84
DERIV2FN.PAS, 84–86
DERIV2.INC, 64, 71–73
DERIV2.PAS, 73–75
Derivative, 63
approximation of, 64–86
DERIVFN.INC, 64, 80–81
DERIVFN.PAS, 81–82
DERIV.INC, 64, 66–68
DERIV.PAS, 68–70
Determinant of a matrix, 105–109
DET.INC, 106–108, 128
DET.PAS, 108–109
Diagonally dominant, 126
Diagonal matrix, 131
Differential equations
first-order, 159–171
coupled, 186–195
linear, 155
nth order, 157, 178–185
ordinary, 155
second-order, 158, 172–177, 192,
208–219
coupled, 196–207
stiff, 161
systems of, 157
Direct factorization of matrices, 106,
120–125
DIRFACT.INC, 106, 120–122
DIRFACT.PAS, 122–125
Distribution disks, 7–11
DIVDIF.INC, 49–50
DIVDIF.PAS, 50–51

E

Eigensystem, 149
Eigenvalue, 131–132
Eigenvector, 131

Enhanced Graphics Adapter, 3, 261, 267
 rebuilding for, 269–270
EXP.LSQ, 225–226, 267

F

Fast Fourier Transform, 233
 algorithms, 241–243
 applications, 244–260
 sample program, 252–260
FFT87B2.INC, 233–235
 algorithms, 241–243
 with the math coprocessor, 270
FFT87B4.INC, 233–235,
 algorithms, 241–243
 and REALFFT.INC, 237
FFTB2.INC, 233–234
 algorithms, 241–243
 rebuilding for FFTIBM.COM, 268
 with the math coprocessor, 270
FFTB4.INC, 233–235
 algorithms, 241–243
 and REALFFT.INC, 237
FFTDEMO.COM, 269
FFTDEMO.PAS, 264–269
FFTHERC.COM, 261
FFTIBM.COM, 261
 graphics demo, 264–266
 rebuilding, 268–269
FFT procedure, 242–243
FFTPROGS.PAS, 252–260
Forward substitution, 120
Forward transform, 234
FOURIER.LSQ, 225, 267
Fourier series, 225
 in graphics programs, 262
Fourier transform, 3, 233–260
 in graphics demo, 264–265

G

GAUSELIM.INC, 106, 114–115
GAUSELIM.PAS, 115–116
Gaussian elimination, 106
 with partial pivoting, 106, 117–119
Gaussian quadrature, 88
 using Legendre polynomials, 98–101
GAUSSIDL.INC, 106, 126–127
GAUSSIDL.PAS, 128–130
Gauss-Jordan elimination, 110
Gauss-Seidel iterative method, 106,
 126–130
GENERIC.LSQ, 267
Goodness of fit, 222

Graphics
 demo programs, 12, 262–267
 printing, 265–267
 system requirements, 3, 261
GRAPHICS.COM, 267
GRAPHICS.EGA, 269

H

Heap/Stack collision, 145, 190, 202
Hercules Monochrome Graphics Adapter,
3, 12, 261
 rebuilding for, 269

I

Initial value problems, 155–157
 Adams-Bashforth/Adams-Moulton,
 168–171
 Runge-Kutta order five, 163–167
 Runge-Kutta order four, 159–167,
 172–207
Installation, 8–11
Integration, 87–104
INTERDRV.INC, 64, 76–77
INTERDRV.PAS, 77–79
Interpolation, 43, 158
 cubic splines, 52–62
 Lagrange polynomials, 45–48
 Newton's divided-difference method,
 49–51
INVERSE.INC, 106, 110–111
Inverse of a matrix, 105–106, 110–113
INVERSE.PAS, 111–113
Inverse power method, 131–132, 137–142
Inverse transform, 234
INVPOWER.INC, 131–132, 137–139
INVPOWER.PAS, 139–142
IOCHECK.INC, 267
Iterative methods, 15
 cyclic Jacobi, 149–153
 Gauss-Seidel, 106

J

JACOBI.INC, 132, 149–151
JACOBI.PAS, 151–153

L

LAGRANGE.INC, 45–46
Lagrange method, 43, 45–48
LAGRANGE.PAS, 46–48
LAGUERRE.INC, 37–38
LAGUERRE.PAS, 39–41

Laguerre's method, 16
 finding roots of complex polynomial,
 37–41
 LEAST.INC, 106, 222–226
 LEAST.MOD, 267
 LEAST.PAS, 227–231
 Least-squares approximation, 222–231
 Least-squares solution
 graphics demo, 262–263
 linear regression, 221
 multiple regression, 221
 Linear equations, 105–106
 differential, 155
 with direct factoring, 120–125
 with Gaussian elimination, 114–119
 LINSHOT2.INC, 156, 158, 215–218
 LINSHOT2.PAS, 218–219
 Lipschitz condition, 157
 LOG.LSQ, 226
 LSQDEMO.COM, 268
 LSQDEMO.PAS, 263
 rebuilding for EGA, 269
 rebuilding for printing, 265–267
 LSQHERC.COM, 261
 rebuilding for Hercules, 269
 LSQIBM.COM, 261–263
 graphics demo, 265–267
 rebuilding, 267
 rebuilding for EGA, 269
 LU_Decompose, 120–121
 LU_Solve, 120–122

M

MakeSinCosTable, 242–243
 Mark/Release, 96, 99, 145
 in Adams-Bashforth/Adams-Moulton,
 170
 in complex fast Fourier, 245–252
 in Linear Shooting/Runge-Kutta, 217
 in Runge-Kutta, 161, 182, 191, 202, 211
 in Runge-Kutta-Fehlberg, 165
 Matrix
 algebra, 105
 diagonal, 131
 direct factorization, 106, 120
 identity, 139
 nonsingular, 106, 120–125
 orthogonal, 134
 permutation, 120
 rotation, 149
 square, 106, 131–133, 137
 symmetric, 132, 149–153

Mesh points, 156
 MULLER.INC, 33–35
 MULLER.PAS, 35–36
 Muller's method, 16
 finding roots of complex function,
 33–36

N

New/Dispose, 96, 99, 145
 in Adams-Bashforth/Adams-Moulton,
 170
 in complex fast Fourier, 245–252
 in Linear Shooting/Runge-Kutta, 217
 in Runge-Kutta, 161, 182, 191, 202, 211
 in Runge-Kutta-Fehlberg, 165
 NEWTDEFL.INC, 28–30
 NEWTDEFL.PAS, 30–32
 Newton-Horner method, 15–16
 with deflation, 28–32
 Newton-Raphson method, 15–16
 root of a function using, 21–24
 Newton's general divided-difference
 algorithm, 43, 49–51
 Nonlinear shooting method, 158,
 208–214
 Numerical differentiation, 63–65
 five-point formulas, 64, 66–75
 three-point formulas, 64, 66–75
 two-point formulas, 64, 66–70
 Nyquist frequency, 239, 264

P

Partial pivoting, 106, 117
 and direct factoring, 120
 PARTPIVT.INC, 106, 117–118
 PARTPIVT.PAS, 118–119
 POLY.LSQ, 224, 267
 Polynomials
 Lagrange, 89
 Legendre, 98–100
 methods to approximate roots of, 16,
 25–41
 POWER.INC, 131–135
 POWER.LSQ, 225, 267
 Power method, 131–136
 and Wielandt's deflation, 143–148
 POWER.PAS, 135–136
 Powers-of-four, 234
 Powers-of-two, 234

R

RAPHSON2.INC, 11
RAPHSON.INC, 9, 21–22
RAPHSON.PAS, 22–24
REALCNVL.INC, 233, 235, 238
 application, 248–249
REALCORR.INC, 233, 235, 238
 application, 251–252
REALFFT.INC, 233, 235, 237
 application, 245–246
Richardson extrapolation
 and numerical integration, 64, 80–82
 and Romberg method, 88, 102–104
numerical integration, 64, 80–82
RKF_1.INC, 156, 163–165
RKF_1.PAS, 165–167
Romberg algorithm, 102–104
ROMBERG.INC, 88, 102–103
Romberg method, 88
 using trapezoidal rule, 102–104
ROMBERG.PAS, 103–104
Roots of an equation, 15–42
Rotation matrix, 149–153
Runge-Kutta-Fehlberg, 156, 163–167
RUNGE_1.INC, 156–157, 159–161
RUNGE_1.PAS, 161–162
RUNGE_2.INC, 157, 172–175
RUNGE_2.PAS, 175–177
Runge-Kutta formulas, 179, 187–188,
 197–199
Runge-Kutta methods, 156
 fifth-order, 163
 fourth-order, 156–157, 159–168,
 172–186
RUNGE_N.INC, 157, 178–182
RUNGE_N.PAS, 182–185
RUNGE_S1.INC, 157, 186–191
RUNGE_S1.PAS, 191–195
RUNGE_S2.INC, 157, 196–203
RUNGE_S2.PAS, 203–207

S

SECANT.INC, 25–26
Secant method, 16
 in nonlinear equations, 158, 208–214
 root of a function using, 25–27
SECANT.PAS, 26–27
SHOOT2.INC, 158, 208–211
SHOOT2.PAS, 211–214
Shooting method, 158
 linear, 215–219
 nonlinear, 208–214

SideKick, 96, 99, 145, 190, 202
SIMPSON.INC, 87, 89–90
SIMPSON.PAS, 90–91
Simpson's method, 87–97
Splines (see Cubic spline methods)
SuperKey, 96, 99, 145, 190, 202
System requirements, 3

T

TestForRoot, 17
TestInput, 241, 243
TNArraySize, 12
TNcomplex, 33–34
TNCompVector, 37–38
TNIntVector, 28–29
TNNearlyZero, 12, 16–17
TNTargetF, 18
TNvector, 12
Trapezoid composite rule, 92–94
Trapezoid method, 87–88
TRAPZOID.INC, 87, 92–93
TRAPZOID.PAS, 93–94
Turbo-87, 3–4, 12
TURBO-87.COM, 267–268, 270
TURBO.COM, 267–268, 270
TURBO.MSG, 267–268
Turbo Pascal, 1–3, 11
 rebuilding with, 267–268
Turbo Pascal Graphix Toolbox, 3, 12
 in graphics demos, 267–268
 rebuilding, 267–269

U

UNPACK.EXE, 7–8
USER.LSQ, 226

W

WIELANDT.INC, 132, 143–145
Wielandt method, 132
 with deflation, 143–148

Borland Software



BORLAND
INTERNATIONAL

4585 Scotts Valley Drive, Scotts Valley, CA 95066

Available at better dealers nationwide.
To order by credit card, call (800) 255-8008; CA (800) 742-1133;
CANADA (800) 237-1136.

SIDEKICK® THE DESKTOP ORGANIZER

Whether you're running WordStar®, Lotus®, dBASE®, or any other program, SideKick puts all these desktop accessories at your fingertips—Instantly!

A full-screen WordStar-like Editor to jot down notes and edit files up to 25 pages long.

A Phone Directory for names, addresses, and telephone numbers. Finding a name or a number is a snap.

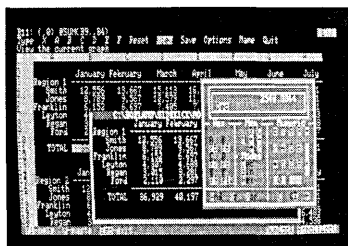
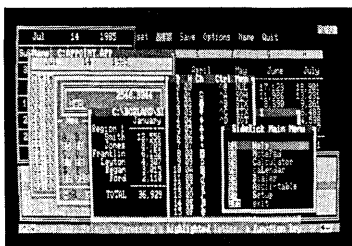
An Autodialer for all your phone calls. It will look up and dial telephone numbers for you. (A modem is required to use this function.)

A Monthly Calendar from 1901 through 2099.

Appointment Calendar to remind you of important meetings and appointments.

A full-featured Calculator ideal for business use. It also performs decimal to hexadecimal to binary conversions.

An ASCII Table for easy reference.



All the SideKick windows stacked up over Lotus 1-2-3.* From bottom to top: SideKick's "Menu Window," ASCII Table, Notepad, Calculator, Appointment Calendar, Monthly Calendar, and Phone Dialer.

Here's SideKick running over Lotus 1-2-3. In the SideKick Notepad you'll notice data that's been imported directly from the Lotus screen. In the upper right you can see the Calculator.

The Critics' Choice

"In a simple, beautiful implementation of WordStar's block copy commands, SideKick can transport all or any part of the display screen (even an area overlaid by the notepad display) to the notepad."

—Charles Petzold, PC MAGAZINE

"SideKick deserves a place in every PC."

—Gary Ray, PC WEEK

"SideKick is by far the best we've seen. It is also the least expensive."

—Ron Mansfield, ENTREPRENEUR

"If you use a PC, get SideKick. You'll soon become dependent on it."

—Jerry Pournelle, BYTE

Suggested Retail Price: \$84.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, PCjr and true compatibles. PC-DOS (MS-DOS) 2.0 or greater. 128K RAM. One disk drive. A Hayes-compatible modem, IBM PCjr internal modem, or AT&T Modem 4000 is required for the autodialer function.



SideKick is a registered trademark of Borland International, Inc. dBASE is a registered trademark of Ashton-Tate. IBM, XT, AT, and PCjr are registered trademarks of International Business Machines Corp. AT&T is a registered trademark of American Telephone & Telegraph Company. Lotus and 1-2-3 are registered trademarks of Lotus Development Corp. WordStar is a registered trademark of MicroPro International Corp. Hayes is a trademark of Hayes Microcomputer Products, Inc. Copyright 1987 Borland International

BOR0060C

Traveling SIDEKICK®

The Organizer For The Computer Age!

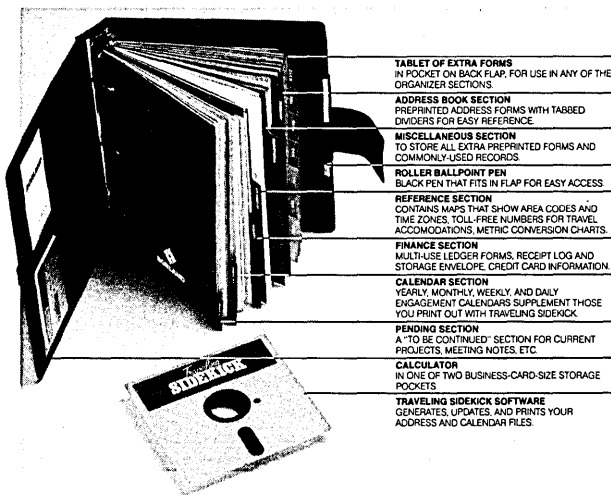
Traveling SideKick is *BinderWare*®, both a binder you take with you when you travel and a software program—which includes a Report Generator—that *generates* and *prints out* all the information you'll need to take with you.

Information like your phone list, your client list, your address book, your calendar, and your appointments. The appointment or calendar files you're already using in your SideKick® can automatically be used by your Traveling SideKick. You don't waste time and effort reentering information that's already there.

One keystroke prints out a form like your address book. No need to change printer paper;

you simply punch three holes, fold and clip the form into your Traveling SideKick binder, and you're on your way. Because Traveling SideKick is CAD (Computer-Age Designed), you don't fool around with low-tech tools like scissors, tape, or staples. And because Traveling SideKick is electronic, it works this year, next year, and all the "next years" after that. Old-fashioned daytime organizers are history in 365 days.

What's inside Traveling SideKick



What the software program and its Report Generator do for you before you go—and when you get back

Before you go:

- Prints out your calendar, appointments, addresses, phone directory, and whatever other information you need from your data files

When you return:

- Lets you quickly and easily enter all the new names you obtained while you were away into your SideKick data files

It can also:

- Sort your address book by contact, zip code or company name
- Print mailing labels
- Print information selectively
- Search files for existing addresses or calendar engagements

Suggested Retail Price: \$69.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, Portable, PCjr, 3270 and true compatibles. PC-DOS (MS-DOS) 2.0 or later. 256K RAM minimum.



BORLAND
INTERNATIONAL

SideKick, BinderWare and Traveling SideKick are registered trademarks of Borland International, Inc. IBM, AT, XT, and PCjr are registered trademarks of International Business Machines Corp. MS-DOS is a registered trademark of Microsoft Corp. Copyright 1987 Borland International

BOR 0083A

SUPERKEY-[®] **THE PRODUCTIVITY BOOSTER**

RAM-resident

Increased productivity for IBM®PCs or compatibles

SuperKey's simple macros are electronic shortcuts to success. By letting you reduce a lengthy paragraph into a single keystroke of your choice, SuperKey eliminates repetition.

SuperKey turns 1,000 keystrokes into 1!

SuperKey can record lengthy keystroke sequences and play them back at the touch of a single key. Instantly. Like magic.

In fact, with SuperKey's simple macros, you can turn "Dear Customer: Thank you for your inquiry. We are pleased to let you know that shipment will be made within 24 hours. Sincerely," into the one keystroke of your choice!

SuperKey keeps your confidential files—confidential!

Without encryption, your files are open secrets. Anyone can walk up to your PC and read your confidential files (tax returns, business plans, customer lists, personal letters, etc.).

With SuperKey you can encrypt any file, *even* while running another program. As long as you keep the password secret, only *you* can decode your file correctly. SuperKey also implements the U.S. government Data Encryption Standard (DES).

- | | |
|---|---|
| <input checked="" type="checkbox"/> RAM resident—accepts new macro files even while running other programs | <input checked="" type="checkbox"/> Keyboard buffer increases 16 character keyboard "type-ahead" buffer to 128 characters |
| <input checked="" type="checkbox"/> Pull-down menus | <input checked="" type="checkbox"/> Real-time delay causes macro playback to pause for specified interval |
| <input checked="" type="checkbox"/> Superfast file encryption | <input checked="" type="checkbox"/> Transparent display macros allow creation of menus on top of application programs |
| <input checked="" type="checkbox"/> Choice of two encryption schemes | <input checked="" type="checkbox"/> Data entry and format control using "fixed" or "variable" fields |
| <input checked="" type="checkbox"/> On-line context-sensitive help | <input checked="" type="checkbox"/> Command stack recalls last 256 characters entered |
| <input checked="" type="checkbox"/> One-finger mode reduces key commands to single keystroke | |
| <input checked="" type="checkbox"/> Screen OFF/ON blanks out and restores screen to protect against "burn in" | |
| <input checked="" type="checkbox"/> Partial or complete reorganization of keyboard | |

Suggested Retail Price: \$99.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, PCjr, and true compatibles: PC-DOS (MS-DOS) 2.0 or greater. 128K RAM. One disk drive.



SuperKey is a registered trademark of Borland International, Inc. IBM, XT, AT, and PCjr are registered trademarks of International Business Machines Corp. MS-DOS is a registered trademark of Microsoft Corp.

BOR 0062C

If you use an IBM® PC, you need

T U R B O
Lightning®

Turbo Lightning teams up with the Random House Concise Word List to check your spelling as you type!

Turbo Lightning, using the 80,000-word Random House Dictionary, checks your spelling as you type. If you misspell a word, it alerts you with a "beep." At the touch of a key, Turbo Lightning opens a window on top of your application program and suggests the correct spelling. Just press one key and the misspelled word is instantly replaced with the correct word.

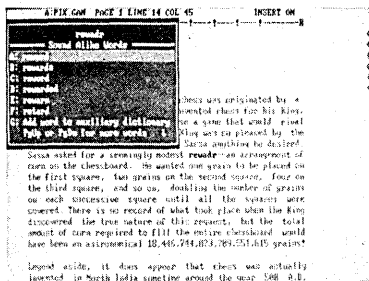
If you ever write a word, think a word, or say a word, you need Turbo Lightning

***You can teach Turbo
Lightning new words***

You can *teach* your new Turbo Lightning your name, business associates' names, street names, addresses, correct capitalizations, and any specialized words you use frequently. Teach Turbo Lightning once, and it knows forever.

***Turbo Lightning is the
engine that powers
Borland's Turbo Lightning
Library®***

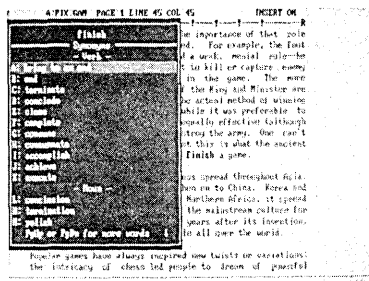
Turbo Lightning brings electronic power to the Random House Concise Word List and Random House Thesaurus. They're at your fingertips—even while you're running other programs. Turbo Lightning will also "drive" soon-to-be-released encyclopedias, extended thesauruses, specialized dictionaries, and many other popular reference works. You get a head start with this first volume in the Turbo Lightning Library.



The Turbo Lightning Proofreader

Turbo Lightning works hand-in-hand with the Random House Thesaurus to give you instant access to synonyms

Turbo Lightning lets you choose just the right word from a list of alternates, so you don't say the same thing the same way every time. Once Turbo Lightning opens the Thesaurus window, you see a list of alternate words; select the word you want, press ENTER and your new word will instantly replace the original word. Pure magic!



The Turbo Lightning Thesaurus

Suggested Retail Price: \$99.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, PCjr, and true compatibles with 2 floppy disk drives. PC-DOS (MS-DOS) 2.0 or greater. 256K RAM. Hard disk recommended.



BORLAND
INTERNATIONAL

Turbo Lightning and Turbo Lightning Library are registered trademarks of Borland International, Inc. IBM, XT, AT, and PCjr are registered trademarks of International Business Machines Corp. Random House is a registered trademark of Random House, Inc. Copyright 1987 Borland International

Copyright 1987 Borland International

BOB 0070B

L I G H T N I N G
WORDWIZARD™

Lightning Word Wizard includes complete, commented Turbo Pascal® source code and all the technical information you'll need to understand and work with Turbo Lightning's "engine."

More than 20 fully documented Turbo Pascal procedures reveal powerful Turbo Lightning engine calls. Harness the full power of the complete and authoritative Random House® Concise Word List and Random House Thesaurus.

Turbo Lightning's "Reference Manual"

Developers can use the versatile on-line examples to harness Turbo Lightning's power to do rapid word searches. Lightning Word Wizard is the forerunner of the database access systems that will incorporate and engineer the Turbo Lightning Library® of electronic reference works.

The ultimate collection of word games and crossword solvers!

The excitement, challenge, competition, and education of four games and three solver utilities—puzzles, scrambles, spell-searches, synonym-seekings, hidden words, crossword solutions, and more. You and your friends (up to four people total) can set the difficulty level and contest the high-speed smarts of Lightning Word Wizard!

Turbo Lightning—Critics' Choice

"Lightning's good enough to make programmers and users cheer, executives of other software companies weep."

Jim Seymour, *PC Week*

"The real future of Lightning clearly lies not with the spelling checker and thesaurus currently included, but with other uses of its powerful look-up engine."

Ted Silveira, *Profiles*

"This newest product from Borland has it all."

Don Roy, *Computing Now!*

Minimum system configuration: IBM PC, XT, AT, PCjr, Portable, and true compatibles. 256K RAM minimum. PC-DOS (MS-DOS) 2.0 or greater. Turbo Lightning software required. Optional—Turbo Pascal 3.0 or greater to edit and compile Turbo Pascal source code.



***Suggested Retail Price: \$69.95
(not copy protected)***

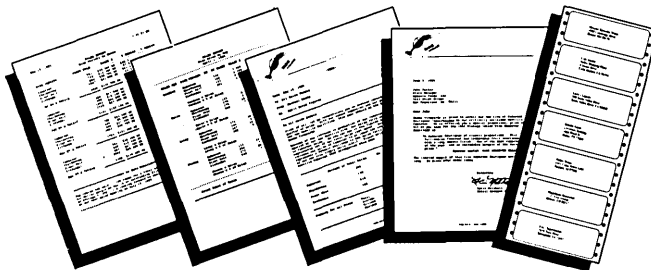
REFLEX[®] THE DATABASE MANAGER

***The high-performance database manager
that's so advanced it's easy to use!***

Lets you organize, analyze and report information faster than ever before! If you manage mailing lists, customer files, or even your company's budgets—Reflex is the database manager for you!

Reflex is the acclaimed, high-performance database manager you've been waiting for. Reflex extends database management with business graphics. Because a picture is often worth a 1000 words, Reflex lets you extract critical information buried in mountains of data. With Reflex, when you look, you see.

The **REPORT VIEW** allows you to generate everything from mailing labels to sophisticated reports. You can use database files created with Reflex or transferred from Lotus 1-2-3,[®] dBASE,[®] PFS: File,[®] and other applications.



Reflex: The Critics' Choice

"... if you use a PC, you should know about Reflex ... may be the best bargain in software today."

Jerry Pournelle, BYTE

"Everyone agrees that Reflex is the best-looking database they've ever seen."

Adam B. Green, InfoWorld

"The next generation of software has officially arrived."

Peter Norton, PC Week

Reflex: don't use your PC without it!

Join hundreds of thousands of enthusiastic Reflex users and experience the power and ease of use of Borland's award-winning Reflex.

Suggested Retail Price: \$149.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, and true compatibles. 384K RAM minimum. IBM Color Graphics Adapter, Hercules Monochrome Graphics Card, or equivalent. PC-DOS (MS-DOS) 2.0 or greater. Hard disk and mouse optional. Lotus 1-2-3, dBASE, or PFS: File optional.



Reflex is a trademark of Borland/Analytica Inc. Lotus 1-2-3 is a registered trademark of Lotus Development Corporation. dBASE is a registered trademark of Ashton-Tate. PFS: File is a registered trademark of Software Publishing Corporation. IBM, XT, AT, and IBM Color Graphics Adapter are registered trademarks of International Business Machines Corporation. Hercules Graphics Card is a trademark of Hercules Computer Technology. MS-DOS is a registered trademark of Microsoft Corp. Copyright 1987 Borland International BOR 0066C

REFLEX: THE WORKSHOP™

Includes 22 "instant templates" covering a broad range of business applications (listed below). Also shows you how to customize databases, graphs, crosstabs, and reports. It's an invaluable analytical tool and an important addition to another one of our best sellers, Reflex: The Database Manager.

Fast-start tutorial examples:

Learn Reflex® as you work with practical business applications. The Reflex Workshop Disk supplies databases and reports large enough to illustrate the power and variety of Reflex features. Instructions in each Reflex Workshop chapter take you through a step-by-step analysis of sample data. You then follow simple steps to adapt the files to your own needs.

22 practical business applications:

Workshop's 22 "instant templates" give you a wide range of analytical tools:

Administration

- Scheduling Appointments
- Planning Conference Facilities
- Managing a Project
- Creating a Mailing System
- Managing Employment Applications

Sales and Marketing

- Researching Store Check Inventory
- Tracking Sales Leads
- Summarizing Sales Trends
- Analyzing Trends

Production and Operations

- Summarizing Repair Turnaround

- Tracking Manufacturing Quality Assurance
- Analyzing Product Costs

Accounting and Financial Planning

- Tracking Petty Cash
- Entering Purchase Orders
- Organizing Outgoing Purchase Orders
- Analyzing Accounts Receivable
- Maintaining Letters of Credit
- Reporting Business Expenses
- Managing Debits and Credits
- Examining Leased Inventory Trends
- Tracking Fixed Assets
- Planning Commercial Real Estate Investment

Whether you're a newcomer learning Reflex basics or an experienced "power user" looking for tips, Reflex: The Workshop will help you quickly become an expert database analyst.

Minimum system configuration: IBM PC, AT, and XT, and true compatibles. PC-DOS (MS-DOS) 2.0 or greater. 384K RAM minimum. Requires Reflex: The Database Manager, and IBM Color Graphics Adapter, Hercules Monochrome Graphics Card or equivalent.



***Suggested Retail Price: \$69.95
(not copy protected)***

Reflex is a registered trademark and Reflex: The Workshop is a trademark of Borland/Analytica, Inc. IBM, AT, and XT are registered trademarks of International Business Machines Corp. Hercules is a trademark of Hercules Computer Technology. MS-DOS is a registered trademark of Microsoft Corp. Copyright 1987 Borland International

BOR 0088B

TURBO PASCAL®

Version 3.0 with 8087 support and BCD reals

Free MicroCalc Spreadsheet With Commented Source Code!

FEATURES:

One-Step Compile: No hunting & fishing expeditions! Turbo finds the errors, takes you to them, lets you correct them, and instantly recompiles. You're off and running in record time.

Built-in Interactive Editor: WordStar®-like easy editing lets you debug quickly.

Automatic Overlays: Fits big programs into small amounts of memory.

MicroCalc: A sample spreadsheet on your disk with ready-to-compile source code.

IBM® PC Version: Supports Turtle Graphics, color, sound, full tree directories, window routines, input/output redirection, and much more.

THE CRITICS' CHOICE:

"Language deal of the century . . . Turbo Pascal: it introduces a new programming environment and runs like magic."

—Jeff Duntemann, *PC Magazine*

"Most Pascal compilers barely fit on a disk, but Turbo Pascal packs an editor, compiler, linker, and run-time library into just 39K bytes of random access memory."

—Dave Garland, *Popular Computing*

"What I think the computer industry is headed for: well-documented, standard, plenty of good features, and a reasonable price."

—Jerry Pournelle, *BYTE*

LOOK AT TURBO NOW!

- ☒ More than 500,000 users worldwide.
- ☒ Turbo Pascal is the de facto industry standard.
- ☒ Turbo Pascal wins PC MAGAZINE'S award for technical excellence.
- ☒ Turbo Pascal named "Most Significant Product of the Year" by PC WEEK.
- ☒ Turbo Pascal 3.0—the fastest Pascal development environment on the planet, period.

Suggested Retail Price: \$99.95; CP/M®-80 version without 8087 and BCD: \$69.95

Features for 16-bit Systems: 8087 math co-processor support for intensive calculations. Binary Coded Decimals (BCD): eliminates round-off error! A *must* for any serious business application.

Minimum system configuration: 128K RAM minimum. Includes 8087 & BCD features for 16-bit MS-DOS 2.0 or later and CP/M-86 1.1 or later. CP/M-80 version 2.2 or later 48K RAM minimum (8087 and BCD features not available). 8087 version requires 8087 or 80287 co-processor.



Turbo Pascal is a registered trademark of Borland International, Inc. CP/M is a registered trademark of Digital Research Inc. IBM is a registered trademark of International Business Machines Corp. MS-DOS is a registered trademark of Microsoft Corp. WordStar is a registered trademark of MicroPro International. Copyright 1987 Borland International BOR 0061B

TURBO PASCAL **TURBO TUTOR[®]**

VERSION 2.0

Learn Pascal From The Folks Who Created The Turbo Pascal[®] Family

Borland International proudly presents Turbo Tutor, the perfect complement to your Turbo Pascal compiler. Turbo Tutor is really for everyone—even if you've never programmed before.

And if you're already proficient, Turbo Tutor can sharpen up the fine points. The manual and program disk focus on the whole spectrum of Turbo Pascal programming techniques.

- **For the Novice:** It gives you a concise history of Pascal, tells you how to write a simple program, and defines the basic programming terms you need to know.
- **Programmer's Guide:** The heart of Turbo Pascal. The manual covers the fine points of every aspect of Turbo Pascal programming: program structure, data types, control structures, procedures and functions, scalar types, arrays, strings, pointers, sets, files, and records.
- **Advanced Concepts:** If you're an expert, you'll love the sections detailing such topics as linked lists, trees, and graphs. You'll also find sample program examples for PC-DOS and MS-DOS.[®]

10,000 lines of commented source code, demonstrations of 20 Turbo Pascal features, multiple-choice quizzes, an interactive on-line tutor, and more!

Turbo Tutor may be the only reference work about Pascal and programming you'll ever need!

Suggested Retail Price: \$39.95 (not copy protected)

Minimum system configuration: Turbo Pascal 3.0. PC-DOS (MS-DOS) 2.0 or later. 192K RAM minimum (CP/M-80 version 2.2 or later: 64K RAM minimum).



Turbo Pascal and Turbo Tutor are registered trademarks of Borland International Inc. CP/M is a registered trademark of Digital Research Inc. MS-DOS is a registered trademark of Microsoft Corp. Copyright 1987 Borland International BOR 0064C

TURBO PASCAL **DATABASE TOOLBOX®**

Is The Perfect Complement To Turbo Pascal®

It contains a complete library of Pascal procedures that allows you to sort and search your data and build powerful database applications. It's another set of tools from Borland that will give even the beginning programmer the expert's edge.

THE TOOLS YOU NEED!

TURBO ACCESS Using B+ trees: The best way to organize and search your data. Makes it possible to access records in a file using key words instead of numbers. Now available with complete source code on disk, ready to be included in your programs.

TURBO SORT: The fastest way to sort data using the QUICKSORT algorithm—the method preferred by knowledgeable professionals. Includes source code.

GINST (General Installation Program): Gets your programs up and running on other terminals. This feature alone will save hours of work and research. Adds tremendous value to all your programs.

GET STARTED RIGHT AWAY—FREE DATABASE!

Included on every Toolbox diskette is the source code to a working database which demonstrates the power and simplicity of our Turbo Access search system. Modify it to suit your individual needs or just compile it and run.

THE CRITICS' CHOICE!

"The tools include a B+ tree search and a sorting system. I've seen stuff like this, but not as well thought out, sell for hundreds of dollars."
—Jerry Pournelle, BYTE MAGAZINE

"The Turbo Database Toolbox is solid enough and useful enough to come recommended."
—Jeff Duntemann, PC TECH JOURNAL

Suggested Retail Price: \$69.95 (not copy protected)

Minimum system configuration: 128K RAM and one disk drive (CP/M-80: 48K). 16-bit systems: Turbo Pascal 2.0 or greater for MS-DOS or PC-DOS 2.0 or greater. Turbo Pascal 2.1 or greater for CP/M-86 1.0 or greater. 8-bit systems: Turbo Pascal 2.0 or greater for CP/M-80 2.2 or greater.



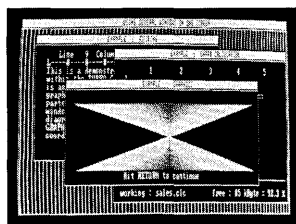
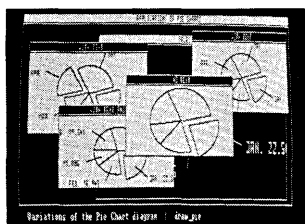
Turbo Pascal and Turbo Database Toolbox are registered trademarks of Borland International Inc. CP/M is a registered trademark of Digital Research, Inc. MS-DOS is a registered trademark of Microsoft Corp. Copyright 1987 Borland International BOR 0063D

TURBO PASCAL **GRAPHIX TOOLBOX®**

A Library of Graphics Routines for Use with Turbo Pascal®

High-resolution graphics for your IBM® PC, AT,® XT,® PCjr,® true PC compatibles, and the Heath Zenith Z-100.™ Comes complete with graphics window management.

Even if you're new to Turbo Pascal programming, the Turbo Pascal Graphix Toolbox will get you started right away. It's a collection of tools that will get you right into the fascinating world of high-resolution business graphics, including graphics window management. You get immediate, satisfying results. And we keep Royalty out of American business because you don't pay any—even if you distribute your own compiled programs that include all or part of the Turbo Pascal Graphix Toolbox procedures.



What you get includes:

- Complete commented source code on disk.
- Tools for drawing simple graphics.
- Tools for drawing complex graphics, including curves with optional smoothing.
- Routines that let you store and restore graphic images to and from disk.
- Tools allowing you to send screen images to Epson®-compatible printers.
- Full graphics window management.
- Two different font styles for graphic labeling.
- Choice of line-drawing styles.
- Routines that will let you quickly plot functions and model experimental data.
- And much, much more . . .

"While most people only talk about low-cost personal computer software, Borland has been doing something about it. And Borland provides good technical support as part of the price."

John Markov & Paul Freiburger, syndicated columnists.

If you ever plan to create Turbo Pascal programs that make use of business graphics or scientific graphics, you need the Turbo Pascal Graphix Toolbox.

Suggested Retail Price: \$69.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, PCjr, true compatibles and the Heath Zenith Z-100. Turbo Pascal 3.0 or later. 192K RAM minimum. Two disk drives and an IBM Color Graphics Adapter (CGA), IBM Enhanced Graphics Adapter (EGA), Hercules Graphics Card or compatible.



Turbo Pascal and Turbo Graphix Toolbox are registered trademarks of Borland International, Inc. IBM, XT, AT, and PCjr are registered trademarks of International Business Machines Corporation. Hercules Graphics Card is a trademark of Hercules Computer Technology. Heath Zenith Z-100 is a trademark of Zenith Data Systems. Epson is a registered trademark of Epson Corp. Copyright 1987 Borland International

BOR 0068C

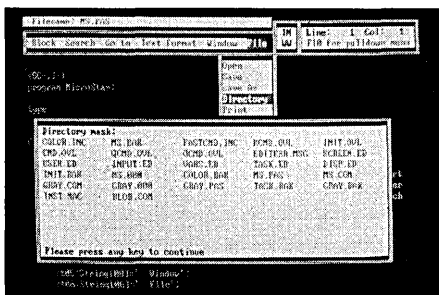
It's All You Need To Build Your Own Text Editor Or Word Processor

Create your own word processor. We provide all the editing routines. You plug in the features you want. You could build a WordStar®-like editor with pull-down menus like Microsoft's® Word, and make it work as fast as WordPerfect®

Simple Editor A complete editor ready to include in your programs. With windows, block commands, and memory-mapped screen routines.

MicroStar A full-blown text editor with a complete pull-down menu user interface, plus a lot more. Modify MicroStar's pull-down menu system and include it in your Turbo Pascal programs.

- Wordwrap
- UN-delete last line
- Auto-indent
- Find and Find/Replace with options
- Set left and right margin
- Block mark, move, and copy
- Tab, insert and overstrike modes, centering, etc.



MicroStar's pull-down menus.

And Turbo Editor Toolbox has features that word processors selling for several hundred dollars can't begin to match. Just to name a few:

- ☒ **RAM-based editor.** You can edit very large files and yet editing is lightning fast.
- ☒ **Memory-mapped screen routines.** Instant paging, scrolling, and text display.
- ☒ **Keyboard installation.** Change control keys from WordStar-like commands to any that you prefer.
- ☒ **Multiple windows.** See and edit up to eight documents—or up to eight parts of the same document—all at the same time.
- ☒ **Multitasking.** Automatically save your text. Plug in a digital clock, an appointment alarm—see how it's done with MicroStar's "background" printing.

Best of all, **source code is included for everything in the Editor Toolbox.**

Suggested Retail Price: \$69.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, 3270, PCjr, and true compatibles. PC-DOS (MS-DOS) 2.0 or greater. 192K RAM. You must be using Turbo Pascal 3.0 for IBM and compatibles.



Turbo Pascal and Turbo Editor Toolbox are registered trademarks of Borland International, Inc. WordStar is a registered trademark of MicroPro International Corp. Word and MS-DOS are registered trademarks of Microsoft Corp. WordPerfect is a trademark of Satellite Software International. IBM, XT, AT, and PCjr are registered trademarks of International Business Machines Corp. BOR 0067B

TURBO PASCAL **GAMEWORKS®**

Secrets And Strategies Of The Masters Are Revealed For The First Time

Explore the world of state-of-the-art computer games with Turbo GameWorks. Using easy-to-understand examples, Turbo GameWorks teaches you techniques to quickly create your own computer games using Turbo Pascal.® Or, for instant excitement, play the three great computer games we've included on disk—compiled and ready to run.

TURBO CHESS

Test your chess-playing skills against your computer challenger. With Turbo GameWorks, you're on your way to becoming a master chess player. Explore the complete Turbo Pascal source code and discover the secrets of Turbo Chess.

"What impressed me the most was the fact that with this program you can become a computer chess analyst. You can add new variations to the program at any time and make the program play stronger and stronger chess. There's no limit to the fun and enjoyment of playing Turbo GameWorks Chess, and most important of all, with this chess program there's no limit to how it can help you improve your game."

***—George Koltanowski, Dean of American Chess, former President of
the United Chess Federation, and syndicated chess columnist.***

TURBO BRIDGE

Now play the world's most popular card game—bridge. Play one-on-one with your computer or against up to three other opponents. With Turbo Pascal source code, you can even program your own bidding or scoring conventions.

"There has never been a bridge program written which plays at the expert level, and the ambitious user will enjoy tackling that challenge, with the format already structured in the program. And for the inexperienced player, the bridge program provides an easy-to-follow format that allows the user to start right out playing. The user can 'play bridge' against real competition without having to gather three other people."

***—Kit Woolsey, writer of several articles and books on bridge,
and twice champion of the Blue Ribbon Pairs.***

TURBO GO-MOKU

Prepare for battle when you challenge your computer to a game of Go-Moku—the exciting strategy game also known as Pente.® In this battle of wits, you and the computer take turns placing X's and O's on a grid of 19×19 squares until five pieces are lined up in a row. Vary the game if you like, using the source code available on your disk.

Suggested Retail Price: \$69.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, Portable, 3270, PCjr, and true compatibles. PC-DOS (MS-DOS) 2.0 or later. 192K RAM minimum. To edit and compile the Turbo Pascal source code, you must be using Turbo Pascal 3.0 for IBM PCs and compatibles.



Turbo Pascal and Turbo GameWorks are registered trademarks of Borland International, Inc. Pente is a registered trademark of Parker Brothers. IBM, XT, AT, and PCjr are registered trademarks of International Business Machines Corporation. MS-DOS is a registered trademark of Microsoft Corporation. Copyright 1987 Borland International
BOR0065C

TURBO PROLOG™

the natural language of Artificial Intelligence

Turbo Prolog brings fifth-generation supercomputer power to your IBM®PC!

STEP-BY-STEP
TUTORIAL AND DEMO PROGRAMS
WITH SOURCE CODE INCLUDED!

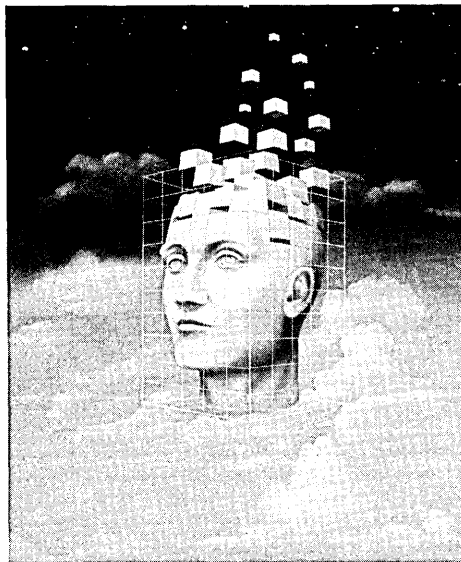
Turbo Prolog takes programming into a new, natural, and logical environment

With Turbo Prolog, because of its natural, logical approach, both people new to programming and professional programmers can build powerful applications such as expert systems, customized knowledge bases, natural language interfaces, and smart information management systems.

Turbo Prolog is a *declarative* language which uses deductive reasoning to solve programming problems.

Turbo Prolog's development system includes:

- A complete Prolog compiler that is a variation of the Clocksin and Mellish Edinburgh standard Prolog.
- A full-screen interactive editor.
- Support for both graphic and text windows.
- All the tools that let you build your own expert systems and AI applications with unprecedented ease.



Turbo Prolog provides a fully integrated programming environment like Borland's Turbo Pascal®, the *de facto* worldwide standard.

You get the complete Turbo Prolog programming system

You get the 200-page manual you're holding, software that includes the lightning-fast Turbo Prolog six-pass

compiler and interactive editor, and the free GeoBase natural query language database, which includes commented source code on disk, ready to compile. (GeoBase is a complete database designed and developed around U.S. geography. You can modify it or use it "as is.")

Minimum system configuration: IBM PC, XT, AT, Portable, 3270, PCjr and true compatibles. PC-DOS (MS-DOS) 2.0 or later. 384K RAM minimum.

**Suggested Retail Price: \$99.95
(not copy protected)**



BORLAND
INTERNATIONAL

Turbo Prolog is a trademark and Turbo Pascal is a registered trademark of Borland International, Inc. IBM, AT, XT, and PCjr are registered trademarks of International Business Machines Corp. MS-DOS is a registered trademark of Microsoft Corp. Copyright 1987 Borland International BOR 0016D

TURBO PROLOG™ TOOLBOX

***Enhances Turbo Prolog with more than 80 tools
and over 8,000 lines of source code***

***Turbo Prolog, the natural language of Artificial Intelligence, is the
most popular AI package in the world with more than 100,000 users.
Our new Turbo Prolog Toolbox extends its possibilities.***

The Turbo Prolog Toolbox enhances Turbo Prolog—our 5th-generation computer programming language that brings supercomputer power to your IBM PC and compatibles—with its more than 80 tools and over 8,000 lines of source code that can be incorporated into your programs, quite easily.

Turbo Prolog Toolbox features include:

- ☒ Business graphics generation: boxes, circles, ellipses, bar charts, pie charts, scaled graphics
- ☒ Complete communications package: supports XModem protocol
- ☒ File transfers from Reflex,* dBASE III,* Lotus 1-2-3,* Symphony*
- ☒ A unique parser generator: construct your own compiler or query language
- ☒ Sophisticated user-interface design tools
- ☒ 40 example programs
- ☒ Easy-to-use screen editor: design your screen layout and I/O
- ☒ Calculated fields definition
- ☒ Over 8,000 lines of source code you can incorporate into your own programs

Suggested Retail Price: \$99.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT or true compatibles. PC-DOS (MS-DOS) 2.0 or later. Requires Turbo Prolog 1.10 or higher. Dual-floppy disk drive or hard disk. 512K.



Turbo Prolog Toolbox and Turbo Prolog are trademarks of Borland International, Inc. Reflex is a registered trademark of Borland/Analytica, Inc. dBASE III is a registered trademark of Ashton-Tate. Lotus 1-2-3 and Symphony are registered trademarks of Lotus Development Corp. IBM, XT, and AT are registered trademarks of International Business Machines Corp. MS-DOS is a registered trademark of Microsoft Corp.

BOR 0240

TURBO BASIC[®]

The high-speed BASIC you've been waiting for!

You probably know us for our Turbo Pascal[®] and Turbo Prolog.[™] Well, we've done it again! We've created Turbo Basic, because BASIC doesn't have to be slow.

If BASIC taught you how to walk, Turbo Basic will teach you how to run!

With Turbo Basic, your only speed is "Full Speed Ahead"! Turbo Basic is a complete development environment with a *lightning fast compiler*, an *interactive editor* and a *trace debugging system*. And because Turbo Basic is also compatible with BASICA, chances are that you already know how to use Turbo Basic.

Turbo Basic ends the basic confusion

There's now one standard: Turbo Basic. And because Turbo Basic is a Borland product, the price is right, the quality is there, and the power is at your fingertips. Turbo Basic is part of the fast-growing Borland family of programming languages we call the "Turbo Family." And hundreds of thousands of users are already using Borland's languages. So, welcome to a whole new generation of smart PC users!

Free spreadsheet included with source code!

Yes, we've included MicroCalc, our sample spreadsheet, complete with source code. So you can get started right away with a "real program." You can compile and run it "as is," or modify it.

A technical look at Turbo Basic

- | | |
|---|---|
| <input checked="" type="checkbox"/> Full recursion supported | executable program, with separate windows for editing, messages, tracing, and execution |
| <input checked="" type="checkbox"/> Standard IEEE floating-point format | |
| <input checked="" type="checkbox"/> Floating-point support, with full 8087 coprocessor integration. Software emulation if no 8087 present | <input checked="" type="checkbox"/> Compile and run-time errors place you in source code where error occurred |
| <input checked="" type="checkbox"/> Program size limited only by available memory (no 64K limitation) | <input checked="" type="checkbox"/> Access to local, static and global variables |
| <input checked="" type="checkbox"/> EGA and CGA support | <input checked="" type="checkbox"/> New long integer (32-bit) data type |
| <input checked="" type="checkbox"/> Full integration of the compiler, editor, and | <input checked="" type="checkbox"/> Full 80-bit precision |
| | <input checked="" type="checkbox"/> Pull-down menus |
| | <input checked="" type="checkbox"/> Full window management |

Suggested Retail Price: \$99.95 (not copy protected)

Minimum system configuration: IBM PC, AT, XT or true compatibles. 256K. One floppy drive. PC-DOS (MS-DOS) 2.0 or later.



Turbo Basic and Turbo Pascal are registered trademarks and Turbo Prolog is a trademark of Borland International, Inc. IBM, AT, and XT are registered trademarks of International Business Machines Corp. MS-DOS is a registered trademark of Microsoft Corp.
Copyright 1987 Borland International

BOR 0265A

TURBO C[®]

Includes free
MicroCalc spreadsheet
with source code

A complete interactive development environment

**With Turbo C, you can expect what only Borland delivers:
Quality, Speed, Power and Price. And with its compilation
speed of more than 7000 lines a minute, Turbo C makes
everything else look like an exercise in slow motion.**

Turbo C: The C compiler for both amateurs and professionals

If you're just beginning and you've "kinda wanted to learn C," now's your chance to do it the easy way. Turbo C's got everything to get you going. If you're already programming in C, switching to Turbo C will considerably increase your productivity and help make your programs both smaller and faster.

Turbo C: a complete interactive development environment

Like Turbo Pascal[®] and Turbo Prolog,[™] Turbo C comes with an interactive editor that will show you syntax errors right in your source code. Developing, debugging, and running a Turbo C program is a snap!

Technical Specifications

- ☒ **Compiler:** One-pass compiler generating native in-line code, linkable object modules and assembler. The object module format is compatible with the PC-DOS linker. Supports small, medium, compact, large, and huge memory model libraries. Can mix models with near and far pointers. Includes floating point emulator (utilizes 8087/80287 if installed).
- ☒ **Interactive Editor:** The system includes a powerful, interactive full-screen text editor. If the compiler detects an error, the editor automatically positions the cursor appropriately in the source code.
- ☒ **Development Environment:** A powerful "Make" is included so that managing Turbo C program development is easy. Borland's fast "Turbo Linker" is also included. Also includes pull-down menus and windows. Can run from the environment or generate an executable file.
- ☒ Links with relocatable object modules created using Borland's Turbo Prolog into a single program.
- ☒ ANSI C compatible.
- ☒ Start-up routine source code included.
- ☒ Both command line and integrated environment versions included.

"Sieve" benchmark (25 iterations)

	Turbo C	Microsoft[®] C	Lattice C
Compile time	3.89	16.37	13.90
Compile and link time	9.94	29.06	27.79
Execution time	5.77	9.51	13.79
Object code size	274	297	301
Price	\$99.95	\$450.00	\$500.00

Benchmark run on a 6 Mhz IBM AT using Turbo C version 1.0 and the Turbo Linker version 1.0; Microsoft C version 4.0 and the MS overlay linker version 3.51; Lattice C version 3.1 and the MS object linker version 3.05.

Suggested Retail Price: \$99.95* (not copy protected)

*Introductory offer good through July 1, 1987.

Minimum system configuration: IBM PC, XT, AT and true compatibles. PC-DOS (MS-DOS) 2.0 or later. One floppy drive. 320K.



BORLAND
INTERNATIONAL

Turbo C and Turbo Pascal are registered trademarks and Turbo Prolog is a trademark of Borland International, Inc. Microsoft C and MS-DOS are registered trademarks of Microsoft Corp. Lattice C is a registered trademark of Lattice, Inc. IBM, XT, and AT are registered trademarks of International Business Machines Corp.

BOR 0243

EUREKA: THE SOLVER™

The solution to your most complex equations—in seconds!

If you're a scientist, engineer, financial analyst, student, teacher, or any other professional working with equations, Eureka: The Solver can do your Algebra, Trigonometry and Calculus problems in a snap.

Eureka also handles maximization and minimization problems, plots functions, generates reports, and saves an incredible amount of time. Even if you're not a computer specialist, Eureka can help you solve your real-world mathematical problems fast, without having to learn numerical approximation techniques. Using Borland's famous pull-down menu design and context-sensitive help screens, Eureka is easy to learn and easy to use—as simple as a hand-held calculator.

X + exp(X) = 10 solved instantly instead of eventually!

Imagine you have to "solve for X," where $X + \exp(X) = 10$, and you don't have Eureka: The Solver. What you do have is a problem, because it's going to take a lot of time guessing at "X." With Eureka, there's no guessing, no dancing in the dark—you get the right answer, right now. (PS: $X = 2.0705799$, and Eureka solved that one in .4 of a second!)

How to use Eureka: The Solver

It's easy.

1. Enter your equation into the full-screen editor
2. Select the "Solve" command
3. Look at the answer
4. You're done

You can then tell Eureka to

- Evaluate your solution
- Plot a graph
- Generate a report, then send the output to your printer, disk file or screen
- Or all of the above

Some of Eureka's key features

You can key in:

- ☑ A formula or formulas
- ☑ A series of equations—and solve for all variables
- ☑ Constraints (like X has to be < or = 2)
- ☑ A function to plot
- ☑ Unit conversions
- ☑ Maximization and minimization problems
- ☑ Interest Rate/Present Value calculations
- ☑ Variables we call "What happens?," like "What happens if I change this variable to 21 and that variable to 27?"

Eureka: The Solver includes

- ☑ A full-screen editor
- ☑ Pull-down menus
- ☑ Context-sensitive Help
- ☑ On-screen calculator
- ☑ Automatic 8087 math co-processor chip support
- ☑ Powerful financial functions
- ☑ Built-in and user-defined math and financial functions
- ☑ Ability to generate reports complete with plots and lists
- ☑ Polynomial finder
- ☑ Inequality solutions

Minimum system configuration: IBM PC, AT, XT, Portable, 3270 and true compatibles. PC-DOS (MS-DOS) 2.0 and later. 384K.

Suggested Retail Price: \$99.95*
(not copy protected)



Eureka: The Solver is a trademark of Borland International, Inc. IBM, AT, and XT are registered trademarks of International Business Machines Corp. MS-DOS is a registered trademark of Microsoft Corp. Copyright 1987 Borland International

*Introductory price expires July 1, 1987

BOR 0221A

SIDEKICK[®] THE DESKTOP ORGANIZER Release 2.0

Macintosh™

The most complete and comprehensive collection of desk accessories available for your Macintosh!

Thousands of users already know that SideKick is the best collection of desk accessories available for the Macintosh. With our new Release 2.0, the best just got better.

We've just added two powerful high-performance tools to SideKick—Outlook™: The Outliner and MacPlan™: The Spreadsheet. They work in perfect harmony with each other and *while* you run other programs!

Outlook: The Outliner

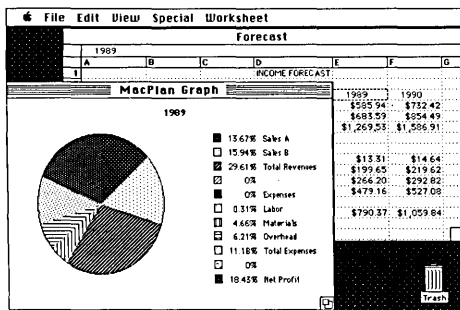
- It's the desk accessory with more power than a stand-alone outliner
- A great desktop publishing tool, Outlook lets you incorporate both text and graphics into your outlines
- Works hand-in-hand with MacPlan
- Allows you to work on several outlines at the same time

MacPlan: The Spreadsheet

- Integrates spreadsheets and graphs
- Does both formulas and straight numbers
- Graph types include bar charts, stacked bar charts, pie charts and line graphs
- Includes 12 example templates free!
- Pastes graphics and data right into Outlook creating professional memos and reports, complete with headers and footers.

SideKick: The Desktop Organizer, Release 2.0 now includes

- ☒ Outlook: The Outliner
- ☒ MacPlan: The Spreadsheet
- ☒ Mini word processor
- ☒ Calendar
- ☒ PhoneLog
- ☒ Analog clock
- ☒ Alarm system
- ☒ Calculator
- ☒ Report generator
- ☒ Telecommunications (new version now supports XModem file transfer protocol)



MacPlan does both spreadsheets and business graphs. Paste them into your Outlook files and generate professional reports.

Suggested Retail Price: \$99.95 (not copy protected)

Minimum system configurations: Macintosh 512K or Macintosh Plus with one disk drive. One 800K or two 400K drives are recommended. With one 400K drive, a limited number of desk accessories will be installable per disk.



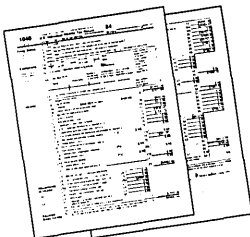
SideKick is a registered trademark and Outlook and MacPlan are trademarks of Borland International, Inc. Macintosh is a trademark of McIntosh Laboratory, Inc. licensed to Apple Computer, Inc. Copyright 1987 Borland International BOR 0069D

REFLEX[®] THE DATABASE MANAGER

The easy-to-use relational database that thinks like a spreadsheet. Reflex for the Mac lets you crunch numbers by entering formulas and link databases by drawing on-screen lines.

5 free ready-to-use templates are included on the examples disk:

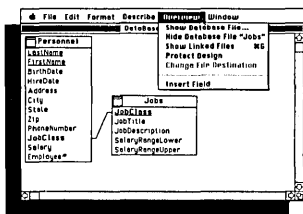
- A sample 1040 tax application with Schedule A, Schedule B, and Schedule D, each contained in a separate report document.
- A portfolio analysis application with linked databases of stock purchases, sales, and dividend payments.
- A checkbook application.



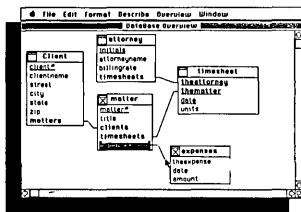
- A client billing application set up for a law office, but easily customized by any professional who bills time.
- A parts explosion application that breaks down an object into its component parts for cost analysis.

Reflex for the Mac accomplishes all of these tasks without programming—using spreadsheet-like formulas. Some other Reflex for the Mac features are:

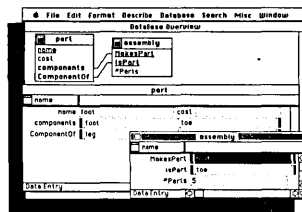
- Visual database design.
- "What you see is what you get" report and form layout with pictures.
- Automatic restructuring of database files when data types are changed, or fields are added and deleted.
- Display formats which include General, Decimal, Scientific, Dollars, Percent.
- Data types which include variable length text, number, integer, automatically incremented sequence number, date, time, and logical.
- Up to 255 fields per record.
- Up to 16 files simultaneously open.
- Up to 16 Mac fonts and styles are selectable for individual fields and labels.



After opening the "Overview" window, you draw link lines between databases directly onto your Macintosh screen.



The link lines you draw establish both visual and electronic relationships between your databases.



You can have multiple windows open simultaneously to view all members of a linked set—which are interactive and truly relational.

Critic's Choice

- "... a powerful relational database ... uses a visual approach to information management." **InfoWorld**
- "... gives you a lot of freedom in report design; you can even import graphics." **A+ Magazine**
- "... bridges the gap between the pretty programs and the power programs." **Stewart Alsop, PC Letter**



BORLAND
INTERNATIONAL

Suggested Retail Price: \$99.95*
(not copy protected)

Minimum system configuration: Macintosh 512K or Macintosh Plus with one disk drive. Second external drive recommended.

Reflex is a registered trademark of Borland/Analytica, Inc. Macintosh is a trademark of McIntosh Laboratory, Inc. and is used with express permission of its owner.
Copyright 1987 Borland International

*Introductory price expires July 1, 1987

BOR0149A

TURBO PASCAL[®] **MACINTOSH™**

The ultimate Pascal development environment

Borland's new Turbo Pascal for the Mac is so incredibly fast that it can compile 1,420 lines of source code in the 7.1 seconds it took you to read this!

And reading the rest of this takes about 5 minutes, which is plenty of time for Turbo Pascal for the Mac to compile at least 60,000 more lines of source code!

Turbo Pascal for the Mac does both Windows and "Units"

The separate compilation of routines offered by Turbo Pascal for the Mac creates modules called "Units," which can be linked to any Turbo Pascal program. This "modular pathway" gives you "pieces" which can then be integrated into larger programs. You get a more efficient use of memory and a reduction in the time it takes to develop large programs.

Turbo Pascal for the Mac is so compatible with Lisa* that they should be living together

Routines from Macintosh Programmer's Workshop Pascal and Inside Macintosh can be compiled and run with only the subtlest changes. Turbo Pascal for the Mac is also compatible with the Hierarchical File System of the Macintosh.

The 27-second Guide to Turbo Pascal for the Mac

- Compilation speed of more than 12,000 lines per minute
- "Unit" structure lets you create programs in modular form
- Multiple editing windows—up to 8 at once
- Compilation options include compiling to disk or memory, or compile and run
- No need to switch between programs to compile or run a program
- Streamlined development and debugging
- Compatibility with Macintosh Programmer's Workshop Pascal (with minimal changes)
- Compatibility with Hierarchical File System of your Mac
- Ability to define default volume and folder names used in compiler directives
- Search and change features in the editor speed up and simplify alteration of routines
- Ability to use all available Macintosh memory without limit
- "Units" included to call all the routines provided by Macintosh Toolbox

Suggested Retail Price: \$99.95* (not copy protected)

*Introductory price expires July 1, 1987

Minimum system configuration: Macintosh 512K or Macintosh Plus with one disk drive.

**3 MacWinners
from Borland!**
First there was SideKick
for the Mac, then Reflex
for the Mac, and now
Turbo Pascal for the Mac!



Turbo Pascal and SideKick are registered trademarks of Borland International, Inc. and Reflex is a registered trademark of Borland/Analytica, Inc. Macintosh is a trademark of McIntosh Laboratories, Inc. licensed to Apple Computer with its express permission. Lisa is a registered trademark of Apple Computer, Inc. Inside Macintosh is a copyright of Apple Computer, Inc.
Copyright 1987 Borland International BOR 0167A

Borland Software ORDER TODAY



BORLAND

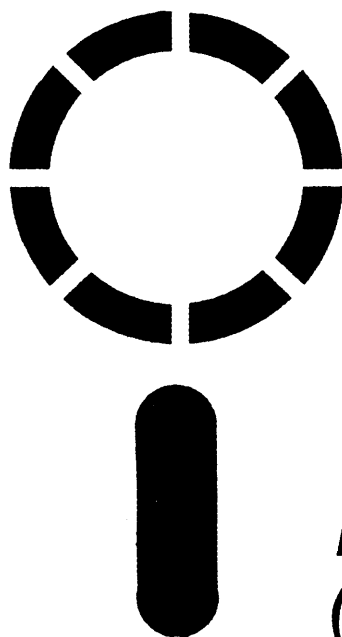
I N T E R N A T I O N A L

4585 Scotts Valley Drive

Scotts Valley, California

95066

***To Order
By Credit
Card,
Call
(800)
255-8008***



***In
California
call
(800)
742-1133***

***In Canada call
(800) 237-1136***

NOTES

NOTES

NOTES

TURBO PASCAL NUMERICAL METHODS TOOLBOX™

*A complete collection
of Turbo Pascal®
routines and programs*

IBM® VERSION
PC, XT, AT, & True Compatibles

New from Borland's Scientific & Engineering Division, Turbo Pascal Numerical Methods Toolbox implements the latest high-level mathematical methods to solve common scientific and engineering problems. Fast.

So every time you need to calculate an integral, work with Fourier Transforms or incorporate any of the classical numerical analysis tools into your programs, you don't have to reinvent the wheel. Because the Numerical Methods Toolbox is a complete collection of Turbo Pascal routines and programs that gives you applied state-of-the-art math tools. It also includes two graphics demo programs, Least Squares Fit and Fast Fourier Transforms, to give you the picture along with the numbers.

The Numerical Methods Toolbox is a must for you if you're involved with any type of scientific or engineering computing. Because it comes with complete source code, you have total control of your application.

What Numerical Methods Toolbox will do for you now:

- Find solutions to equations
- Interpolations
- Calculus: numerical derivatives and integrals
- Matrix operations: inversions, determinants and eigenvalues
- Differential equations

- Least squares approximations
- Fourier transforms

5 free ways to look at "Least Squares Fit"!

As well as a free demo "Fast Fourier Transforms," you also get "Least Squares Fit" in 5 different forms—which gives you 5 different methods of fitting curves to a collection of data points. You instantly get the picture! The 5 different forms are:

1. Power
2. Exponential
3. Logarithm
4. 5-term Fourier
5. 5-term Polynomial

They're all ready to compile and run "as is." To modify or add graphics to your own programs, you simply add Turbo Graphix Toolbox® to your software library. Our Numerical Methods Toolbox is designed to work hand-in-hand with our Turbo Graphix Toolbox to make professional graphics in your own programs an instant part of the picture!

Minimum system configuration: IBM PC, XT, AT and true compatibles. PC-DOS (MS-DOS) 2.0 or later. 256K. Turbo Pascal 2.0 or later. The graphics modules require a graphics monitor with an IBM CGA, IBM EGA, or Hercules compatible adapter card, and require the Turbo Graphix Toolbox. MS-DOS generic version will not support Turbo Graphix Toolbox routines. An 8087 or 80287 numeric co-processor is not required, but recommended for optimal performance.

Turbo Pascal Numerical Methods Toolbox is a trademark and Turbo Pascal and Turbo Graphix Toolbox are registered trademarks of Borland International, Inc. IBM, XT, and AT are registered trademarks of International Business Machines Corp. MS-DOS is a registered trademark of Microsoft Corp. Hercules is a trademark of Hercules Computer Technology. Apple is a registered trademark of Apple Computer, Inc. Macintosh is a trademark of McIntosh Laboratory, Inc. licensed to Apple Computer.

Copyright 1986 Borland International BOR 0224



BORLAND
INTERNATIONAL

4585 SCOTTS VALLEY DRIVE
SCOTTS VALLEY, CA 95066

ISBN 0-87524-157-3